

EPREUVE OPTIONNELLE d'INFORMATIQUE CORRIGE

Question 1 : Un câble de catégorie 5 fait référence a :

- A – un câble coaxial fin
- B – un câble coaxial épais
- C – une paire torsadée non blindée (UTP)
- D – une paire torsadée blindée (STP)

Question 2 : Dans l'alphabet CCITT n° 5 les symboles sont codés sur:

- A – 5 bits
- B – 7 bits
- C – 8 bits
- D – variable

Question 3 : En langage C parmi les identificateurs suivants, quels sont ceux qui sont invalides :

- A – \$ un-prix
- B – une-somme
- C – -une-somme
- D – 1prix

Question 4 : A quelle valeur hexadécimale correspond la valeur décimale 19 ? :

- A – 9
- B – 13
- C – 31
- D – 16

Question 5 : Que signifie le sigle VPN ? :

- A – Virtual Permanent Network
- B – Voie Privée Numérique
- C – Virtual Private Network
- D – Voice Private Node

Question 6 : Une cellule ATM a une taille de :

- A – 32 octets
- B – 53 octets
- C – 64 octets
- D – 128 octets

Question 7 : Le code de HUFFMAN permet de :

- A – crypter les données
- B – contrôler les erreurs
- C – compresser les données
- D – calculer le chemin le plus court

Question 8 : parmi les langages suivants, lequel est orienté « objet » ? :

- A – Pascal
- B – C
- C – C++
- D – JAVA

Question 9 : Un FIREWALL est :

- A – une passerelle d’interconnexion de réseaux
- B – une machine permettant de sécuriser un réseau
- C – un serveur de fichier
- D – un site Web

Question 10 : Identifier l’adresse MAC (Medium Access Control) parmi les adresses suivantes :

- A – 127
- B – 193.16.160.0
- C – 00 : 80 : e4 : 00 : 09 : f2
- D – FF.FF.FF.00

Question 11 : Ethernet est un réseau :

- A – local
- B – métropolitain
- C – distant
- D – virtuel

Question 12 : Quelle est l’unité de mesure de la rapidité de modulation ? :

- A – bit/sec
- B – baud
- C – hertz
- D – décibel

Question 13 : SYBASE est un :

- A – système d’exploitation de base
- B – système de gestion de base de données
- C – langage de programmation
- D – nom de serveur

Question 14 : Quelles sont les protocoles de niveau liaison (ligne) utilisés pour accéder à l’Internet en passant par le réseau RTC ? :

- A – TCP
- B – SLIP
- C – IP
- D – PPP

Question 15 : Dans le système Unix, dans votre répertoire de base se trouve un fichier appelé « .Profile ». A quoi sert-il ? :

- A – il est exécuté automatiquement lors de la phase de login
- B – il contient diverses initialisations
- C – il décrit le profil des utilisateurs
- D – il filtre les paquets

Question 16 : Donnez le nom de l’organisme international à l’origine de la recommandation X.25 :

- A – AFNOR
- B – UIT-T
- C – ISO
- D – IEEE

Question 17 : Il existe 2 technologies de commutation, lesquelles ? :

- A – la commutation spatiale
- B – la commutation temporelle
- C – la commutation de cellules

D – la commutation de circuits

Question 18 : Parmi les commandes suivantes, lesquelles font de l'émulation de terminal :

- A – FTP
- B – TELNET
- C – RLOGIN
- D – NFS

Question 19 : Lorsque le transfert a lieu sous contrainte temporelle, on parle de transfert :

- A – synchrone
- B – asynchrone
- C – isochrone
- D – symétrique

Question 20 : Internet est un réseau :

- A – privé
- B – public
- C – local
- D – interne

Question 21 : A quoi sert la technologie FLASH ? :

- A – dupliquer le contenu d'un disque
- B – accélérer la vitesse d'un processeur
- C – créer des sites Web
- D – concurrencer HTML

Question 22 : Pour transporter la voix numérisée non compressée, il faut un canal à :

- A – 16 Kb/s
- B – 32 Kb/s
- C – 64 Kb/s
- D – 128 Kb/s

Question 23 : Quel est l'intérêt d'un code de longueur variable ? :

- A – crypter les données
- B – calculer le CRC
- C – compresser les données
- D – calculer la taille d'un fichier

Question 24 : A quoi correspond la valence d'un signal ? :

- A – au nombre de bits par état
- B – à la largeur de la bande passante
- C – au nombre d'états que peut prendre le signal
- D – à la valeur du rapport signal sur bruit

Question 25 : Que fait le programme ci-dessous ? :

```
INPUT P
F = 1
FOR N = 1 TO P
  F = F * N
  PRINT F
NEXT N
END
```

- A – remplit un tableau de N cases
- B – calcule la factorielle de N
- C – imprime un message d'erreur

REMARQUE :

Les programmes en Pascal et C sont des exemples et peuvent être discutés en terme d'implémentation et de construction. Le choix qui a été fait, est celui d'une découpe procédurale importante.

Cela permet de mieux faire ressortir un algorithme principal.

Ensuite chaque tâche est détaillée dans la procédure ou fonction correspondante.

D'autre part, le programme Pascal du premier exercice n'est pas optimisé, et se présente comme une Traduction quasi-directe du programme C.

B) CHEMIN**Le principe est le suivant :**

On part des coordonnées (0,0), puis on trace une droite vers (100,100). Si la droite traverse un carré, on regarde si l'intersection se trouve en X ou en Y, selon le cas, on stocke, respectivement, dans un arbre le chemin les coordonnées du coin inférieur droit ou les coordonnées du coin supérieur droit de ce carré. On recommence à partir de ces coordonnées à tracer une droite vers (100,100) et ainsi de suite. Pour finir on extrait de l'arbre le chemin optimal pour alimenter le fichier (CHEMIN.OUT).

Programme Pascal (TP7)

```

Program Chemin ;
Uses Crt;
Const
    COTE =5

Type
    PPoint = ^Point ;
    Point = Record
        x, y : integer ;
        suiv : PPoint ;
    end ;

    Pcarre = ^Carre ;
    Carre = Record
        Csg,cid : Point ;
        Cote : integer ;
        Suiv : Pcarre ;
    End ;

    PNoeud = ^Noeud ;
    Noeud = Record
        P : Point ;
        V : Real ;
        ga, dr : Pnoeud ;
    end ;

    Parc = Record
        V : Real ;
        Parcours, liste, cour : PPoint ;
    End ;

Var
    N : integer;
    Arrive : Point;
    Lcarre : PCarre;
    Arbre : PNoeud;
    Sortie : TEXT;

```

```

function creer_sortie : Boolean;
Begin
  Assigne(sortie, "chemin.out") ;
  {$I-}
  Rewrite(sortie) ;
  {$I+}
  créer_sortie :=IOResult<>0 ;
End ;

(* verifie que les carrés soient distants d'au moins une unité *)
function carre_chevauche(lcar, car : PCarre ) :Boolean ;
Var
  cour : PCarre;
Begin
  cour := lcar;
  while (cour<>car) do
  begin
    if (( car^.csg.x - COTE-1 < cour^.csg.x) and (cour^.csg.x <
car^.cid.x + 1) and
      (car^.cid.y -1 < cour^.csg.y) and (cour^.csg.y < car^.csg.y +
COTE+1)) then
      begin
        Carre_chevauche :=True ;
        Exit ;
      End ;
      cour := cour^.suiv;
    }
    Carre_chevauche :=False ;
  End ;

(* initialisation des données*)
function init :boolean ;
Var
  Entree : TEXT;
  nouv, mem : PCarre;
  count : integer;
Begin
  Assign(Entree,"chemin.in");
  {$I-}
  Reset(Entree);
  {$I+}
  if (IOResult<>0) then
  begin
    Readln(entree,N);
    if ((N<0)or(N>30)) then
    begin
      writeln(sortie,"Valeur d'entree hors-limite");
      init :=False ;
      exit ;
    end ;
    New(lcarre);
    Lcarre^.cote := COTE;
    Readln(entree, lcarre^.csg.x, lcarre^.cid.y);
    if ((lcarre^.cid.y<0) or (lcarre^.cid.y>95) or (lcarre^.csg.x<0) or (lcarre^.csg.x>95))
then
  begin
    writeln(sortie,"Valeur d'entree hors-limite");
    init :=False ;
    exit ;
  end ;

```

```

lcarre^.csg.y := lcarre^.cid.y + COTE;
lcarre^.cid.x := lcarre^.csg.x + COTE;
lcarre^.suiv := NIL;
mem := lcarre;
count := 1;
while (count < N) do
begin
  New(nouv);
  Mem^.suiv := nouv;
  Nouv^.cote := COTE;
  Readln(entree, nouv^.csg.x, nouv^.cid.y);
  if ((nouv^.cid.y<0) or (nouv^.cid.y>95) or (nouv^.csg.x<0)
or (nouv^.csg.x>95)) then
begin
  writeln(sortie,"Valeur d'entree hors-limite");
  init :=False ;
  exit ;
end ;
nouv^.csg.y := nouv^.cid.y + COTE;
nouv^.cid.x := nouv^.csg.x + COTE;
nouv^.suiv := NIL;
if (carre_chevauche(lcarre,nouv)) then
begin
  writeln(sortie,"Des carrés se chevauchent");
  init :=False ;
  exit ;
end ;
mem := nouv;
inc(count);
end ;
close(entree);
init :=True;
end
else begin
  writeln(sortie,"Impossible d'ouvrir le fichier d'entree");
  init :=False ;
end ;
end ;

```

(* si la droite (p1 p2) passe par un carre, renvoie ce carre, sinon NIL *)

```

function passe_par(p1, p2 :Point) :Pcarre ;
{
  a, b : real;
  car : PCarre;
  p : Point;

  a := (p2.y - p1.y)/(p2.x - p1.x);
  b := p1.y - a*p1.x;
  for p.x:=p1 TO p2.x DO
  Begin
    p.y := round(a*p.x+b);
    car := lcarre;
    while (car<>NIL) do
    begin
      (* Si coord du trajet passe par carre...*)
      if ((p.x > car^.csg.x) and (p.x < car^.cid.x) and
      (p.y > car^.cid.y) and (p.y < car^.csg.y)) then
      (* Et si carre != pt de depart *)
      if (Not(((p1.x = car^.csg.x) and (p1.y = car^.csg.y)) or
      ((p1.x = car^.cid.x) and (p1.y = car^.cid.y))))

```

then

```

(* Et si carre <> pt d'arrive *)
if (Not(((p2.x = car^.csg.x) and (p2.y = car-
>csg.y)) or
        ((p2.x = car^.cid.x) and (p2.y = car^.cid.y))))
then
begin
    passe_par :=car;
    exit ;
end ;
    car:=car^.suiv;
end ;
    Inc(p.x) ;
End ;
Passe_par :=NIL;
End ;

```

```

(* création de l'arbre des déplacements possibles *)
creer_arbre(a :PNoeud ; dest, destprec :Point) :Pnoeud ;
Var
    obstacle :PCarre;
    nouv :Pnoeud;

```

```

Begin
    if (a=NIL) then
    begin
        New(a);
        With a^ do
        begin
            p.x := 0;
            p.y := 0;
            p.suiv := NIL;
            v := 0;
            dr := NIL;
            ga := NIL;
        end ;
    end ;
    obstacle := passe_par(a^.p,dest);
    if (obstacle<>NIL) then
    begin
        if (passe_par(a^.p,obstacle^.csg)<>NIL) then
        begin
            New(a^.ga);
            A^.ga^.p := a^.p;
            A^.ga^.v = -1;
            Créer_arbre :=creer_arbre(a^.ga,obstacle^.csg,dest);
        end
        else
            if (obstacle^.csg.x <> a^.p.x) then
            begin
                New(a^.ga);
                A^.ga^.p.x := a^.p.x;
                A^.ga^.p.y := a^.p.y;
                Créer_arbre :=creer_arbre(a^.ga,obstacle^.csg,destprec);
            end
            else
                a^.ga := NIL;
            end
        end
    end
    if (passe_par(a^.p,obstacle^.cid)<>NIL) then
    begin
        New(a^.dr);
        A^.dr^.p := a^.p;
    end
end

```

```

        A^.dr^.v = -1;
        Créer_arbre :=créer_arbre(a^.gdr,obstacle^.cid,dest);
    end
    else
        if (obstacle^.cid.x <> a^.p.x) then
            begin
                New(a^.dr);
                A^.dr^.p.x := a^.p.x;
                A^.dr^.p.y := a^.p.y;
                Créer_arbre :=créer_arbre(a^.dr,obstacle^.cid,destprec);
            end
            else
                a^.dr := NIL;
        end
    end
else begin
    if ((dest.x = arrive.x) and (dest.y = arrive.y)) then
        begin
            New(nouv);
            Nouv^.p.x := dest.x;
            Nouv^.p.y := dest.y;
            Nouv^.v := sqrt((dest.x-a^.p.x)*(dest.x-a^.p.x)+(dest.y-
            a^.p.y)*(dest.y-a^.p.y));
            Nouv^.p.suiv := NIL;
            Nouv^.dr = NIL;
            Nouv^.ga = NIL;
            A^.dr = nouv;
            A^.ga = NIL;
        end
        else begin
            a^.v = sqrt((dest.x-a^.p.x)*(dest.x-a^.p.x)+(dest.y-
            a^.p.y)*(dest.y-a^.p.y));
            a^.p.x := dest.x;
            a^.p.y := dest.y;
            a^.p.suiv = NIL;
            a^.dr = NIL;
            a^.ga = NIL;
            créer_arbre :=créer_arbre(a,destprec,arrive);
        end ;
    end ;
    créer_arbre :=a;
end ;

function copie_liste(l :PPoint) :Ppoint ;
Var
    cop, cour, copcour :PPoint,

Begin
    New(cop);
    Cop^.x := l^.x;
    Cop^.y := l^.y;
    Cop^.suiv := NIL;
    copcour := cop;
    cour := l;
    while (cour^.suiv<>NIL) do
        begin
            with copcour^do
                begin
                    New(suiv);
                    Suiv^.x := cour^.suiv^.x;
                    Suiv^.y := cour^.suiv^.y;
                end
            end
        end
    end
end ;

```

```

        Suiv^.suiv := NIL;
    end ;
    copcour:=copcour^.suiv ;
    cour:=cour^.suiv;
end ;
copie_liste :=cop;
end ;

function efface_liste(p :PPoint) :Ppoint ;
{
    temp :PPoint;

    while (p<>NIL) do
    begin
        temp := p^.suiv;
        dispose(p);
        p:=temp;
    end ;
    efface_liste :=NIL;
end ;

(* Restitue le parcours optimalà partir de l'arbre de déplacement *)
procedure plus_petit_parcours(n :Pnoeud ;s :real ; VAR p :Parc)
Var
    T, Prec :PPoint;
begin
    if (n^.val<>-1) then
    begin
        s:=s+n^.val;
        if (p.liste=NIL) then
        begin
            with p do
            begin
                New(liste);
                Liste^.x := n^.p.x;
                Liste^.y := n^.p.y;
                Liste^.suiv := NIL;
                cour = liste;
            end ;
            prec := NIL;
        end
        else
        begin
            with p.cour do
            begin
                New(suiv);
                Suiv^.x := n^.p.x;
                Suiv^.y := n^.p.y;
                Suiv^.suiv := NIL;
                prec := cour;
            end ;
            p.cour := p.cour^.suiv;
        end ;

        if ((n^.dr<>NIL) or (n^.ga<>NIL)) then
        begin
            if (n^.dr<>NIL) then
                plus_petit_parcours(n^.dr,s,p);
            if (n^.ga<>NIL) then
                plus_petit_parcours(n^.ga,s,p);
            s :=s - n^.v;
        end ;
    end ;
end ;

```

```

        dispose(p.cour);
        if (prec<>NIL) then
            prec^.suiv := NIL;
        p.cour := prec;
    end
else begin
    if ((s<p.v) or (p.v=0)) then
        begin
            p.v :=s;
            T :=efface_liste(p.parcours);
            P.parcours:=copie_liste(p.liste);
        End ;
        s:=s-n^.val;
        dispose(p.cour);
        prec^.suiv :=NIL;
        p.cour:=prec;
    end ;
end
else begin
    if (n^.dr<>NIL) then
        plus_petit_parcours(n^.dr,s,p);
    if (n^.ga) then
        plus_petit_parcours(n^.ga,s,p);
    end ;
end ;
end ;

```

```
function suppr_carre(carre *car) :PCarre
```

```
Var
```

```
    Temp :PCarre;
```

```
Begin
```

```
    while (car<>NIL) do
```

```
        begin
```

```
            temp := car^.suiv;
```

```
            dispose(car);
```

```
            car := temp;
```

```
        end ;
```

```
        suppr_carre :=NIL;
```

```
end ;
```

```
function supprime_arbre(noeud *a) :PNoeud
```

```
begin
```

```
    if (a^.dr) then a^.dr:=supprime_arbre(a^.dr);
```

```
    if (a^.ga) then a^.ga:=supprime_arbre(a^.ga);
```

```
    dispose(a);
```

```
    supprime_arbre :=NIL);
```

```
end ;
```

```
Var
```

```
    p :PPoint;
```

```
    pa : Parc;
```

```
Begin (*PRINCIPAL *)
```

```
    If creer_sortie then
```

```
        begin
```

```
            if init then
```

```
                begin
```

```
                    arrive.x := 100;
```

```
                    arrive.y := 100;
```

```
                    pa.v := 0;
```

```

    pa.liste := NIL;
    pa.parcours := NIL;
    arbre := creer_arbre(NIL, arrive, arrive);
    plus_petit_parcours(arbre, 0, pa);
    lcarre := suppr_carre(lcarre);
    arbre := supprime_arbre(arbre);

    (* Ecriture du résultat dans les fichier CHEMIN.OUT *)
    p := pa.parcours;
    while(p<>NIL) do
    begin
        writeln(sortie, p^.x, ' ', p^.y);
        p := p^.suiv;
    end ;
    end ;
    close(sortie);
end ;
end.

```

Programme C (CC)

```

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

#define COTE 5

// Types

typedef struct pt
{
    int x, y;
    struct pt *suiv;
}point;

typedef struct ca
{
    point csg, cid;
    int cote;
    struct ca *suiv;
}carre;

typedef struct nd
{
    point p;
    float val;
    struct nd *ga, *dr;
}noeud;

struct parc
{
    float val;
    point *parcours, *liste, *cour;
};

// Variables

int N;
point arrive;

```

```

carre *lcarre;
noeud *arbre;
FILE *sortie;

// Fonctions

void creer_sortie()
{
    sortie = fopen("chemin.out","w");
}

// verifie que les carrés soient distants d'au moins une unité
int carre_chevauche(carre *lcar, carre *car)
{
    carre *cour;

    cour = lcar;
    while (cour != car)
    {
        if (( car->csg.x - COTE-1 < cour->csg.x)&&(cour->csg.x < car-
>cid.x + 1)&&
            (car->cid.y -1 < cour->csg.y)&&(cour->csg.y < car->csg.y +
COTE+1))
            return(1);
        cour = cour->suiv;
    }
    return(0);
}

// Initialise les données
int init()
{
    FILE *entree;
    carre *nouv, *mem;
    int count;

    entree = fopen("chemin.in","r");
    if (entree != NULL)
    {
        fscanf(entree,"%d",&N);
        if ((N<0)|(N>30))
        {
            printf("Valeur d'entree hors-limite\n");
            return(0);
        }
        lcarre = malloc(sizeof(carre));
        lcarre->cote = COTE;
        fscanf(entree,"%d",&lcarre->csg.x);
        fscanf(entree,"%d",&lcarre->cid.y);
        if ((lcarre->cid.y<0)|(lcarre->cid.y>95)|(lcarre-
>csg.x<0)|(lcarre->csg.x>95))
        {
            fprintf(sortie,"Valeur d'entree hors-limite\n");
            return(0);
        }
        lcarre->csg.y = lcarre->cid.y + COTE;
        lcarre->cid.x = lcarre->csg.x + COTE;
        lcarre->suiv = NULL;
        mem = lcarre;
        count = 1;
    }
}

```

```

while (count < N)
{
    nouv = malloc(sizeof(carre));
    mem->suiv = nouv;
    nouv->cote = COTE;
    fscanf(entree, "%d", &nouv->csg.x);
    fscanf(entree, "%d", &nouv->cid.y);
    if ((nouv->cid.y < 0) | (nouv->cid.y > 95) | (nouv->csg.x < 0) | (nouv->csg.x > 95))
    {
        fprintf(sortie, "Valeur d'entree hors-limite\n");
        return(0);
    }
    nouv->csg.y = nouv->cid.y + COTE;
    nouv->cid.x = nouv->csg.x + COTE;
    nouv->suiv = NULL;
    if (carre_chevauche(lcarre, nouv))
    {
        fprintf(sortie, "Les carrés ne sont pas assez espacés (%d
%d)\n",
                nouv->csg.x, nouv->cid.y);
        return(0);
    }
    mem = nouv;
    count++;
}
fclose(entree);
return(1);
}
fprintf(sortie, "Impossible d'ouvrir le fichier d'entree\n");
return(0);
}

int round(float f)
{
    float p, g;

    p = (float)floor(f);
    g = (float)ceil(f);
    if ((g-f) > (f-p))
        return((int)p);
    return((int)g);
}

// si la droite (p1 p2) passe par un carre, renvoie ce carre, NULL sinon
carre *passe_par(point p1, point p2)
{
    float a, b;
    carre *car;
    point p;

    a = (float)(p2.y - p1.y)/(float)(p2.x - p1.x);
    b = p1.y - a*p1.x;

    for(p.x=p1.x; p.x<=p2.x; p.x++)
    {
        p.y = round(a * (float)p.x + b);

        car = lcarre;
        while (car)
        {

```

```

        if ((p.x > car->csg.x)&&(p.x < car->cid.x)&&
            (p.y > car->cid.y)&&(p.y < car->csg.y))
            // Si coord du trajet passe par carre...
            if (!((p1.x == car->csg.x)&&(p1.y == car->csg.y))|
                ((p1.x == car->cid.x)&&(p1.y == car->cid.y))))
                // Et si carre != pt de depart
                if (!((p2.x == car->csg.x)&&(p2.y == car->csg.y))|
                    ((p2.x == car->cid.x)&&(p2.y == car->cid.y))))
                    // Et si carre != pt d'arrive
                    return(car);
            car = car->suiv;
    }
}
return(NULL);
}

```

// crée l'arbre des déplacements possibles

```
noeud *creer_arbre(noeud * a, point dest, point destprec)
```

```

{
    carre *obstacle;
    noeud *nou;

    if (!a)
    {
        a = malloc(sizeof(noeud));
        a->p.x = 0;
        a->p.y = 0;
        a->p.suiv = NULL;
        a->val = 0.0f;
        a->dr = NULL;
        a->ga = NULL;
    }
    obstacle = passe_par(a->p,dest);
    if (obstacle)
    {
        if (passe_par(a->p,obstacle->csg))
        {
            a->ga = malloc(sizeof(noeud));
            a->ga->p = a->p;
            a->ga->val = -1;
            creer_arbre(a->ga,obstacle->csg,dest);
        }
        else
            if (obstacle->csg.x != a->p.x)
            {
                a->ga = malloc(sizeof(noeud));
                a->ga->p.x = a->p.x;
                a->ga->p.y = a->p.y;
                creer_arbre(a->ga,obstacle->csg,destprec);
            }
            else
                a->ga = NULL;

        if (passe_par(a->p,obstacle->cid))
        {
            a->dr = malloc(sizeof(noeud));
            a->dr->p = a->p;
            a->dr->val = -1;
            creer_arbre(a->dr,obstacle->cid,dest);
        }
        else
    }
}

```

```

        if (obstacle->cid.x != a->p.x)
        {
            a->dr = malloc(sizeof(noead));
            a->dr->p.x = a->p.x;
            a->dr->p.y = a->p.y;
            creer_arbre(a->dr, obstacle->cid, destprec);
        }
        else
            a->dr = NULL;
    }
else
{
    if ((dest.x == arrive.x)&&(dest.y == arrive.y))
    {
        nouv = malloc(sizeof(noead));
        nouv->p.x = dest.x;
        nouv->p.y = dest.y;
        nouv->val = (float)sqrt((dest.x - a->p.x)*(dest.x - a->p.x) +
            (dest.y - a->p.y)*(dest.y - a->p.y));
        nouv->p.suiv = NULL;
        nouv->dr = NULL;
        nouv->ga = NULL;
        a->dr = nouv;
        a->ga = NULL;
    }
    else
    {
        a->val = (float)sqrt((dest.x - a->p.x)*(dest.x - a->p.x) +
            (dest.y - a->p.y)*(dest.y - a->p.y));
        a->p.x = dest.x;
        a->p.y = dest.y;
        a->p.suiv = NULL;
        a->dr = NULL;
        a->ga = NULL;
        creer_arbre(a, destprec, arrive);
    }
}
return(a);
}

```

```

point *copie_liste(point *l)
{
    point *cop, *cour,
        *copcour;

    cop = malloc(sizeof(point));
    cop->x = l->x;
    cop->y = l->y;
    cop->suiv = NULL;
    copcour = cop;
    cour = l;
    while (cour->suiv)
    {
        copcour->suiv = malloc(sizeof(point));
        copcour->suiv->x = cour->suiv->x;
        copcour->suiv->y = cour->suiv->y;
        copcour->suiv->suiv = NULL;
        copcour = copcour->suiv;
        cour = cour->suiv;
    }
}

```

```

    }
    return(cop);
}

```

```

point *efface_liste(point *p)
{
    point *temp;

    while (p != NULL)
    {
        temp = p->suiv;
        free(p);
        p = temp;
    }
    return(NULL);
}

```

// determine le plus petit parcours à partir de l'arbre de déplacement

```

void plus_petit_parcours(noeud *n, float s, struct parc *p)
{

```

```

    point *prec;

```

```

    if (n->val != -1)
    {

```

```

        s += n->val;

```

```

        if (p->liste == NULL)
        {

```

```

            p->liste = malloc(sizeof(point));

```

```

            p->liste->x = n->p.x;

```

```

            p->liste->y = n->p.y;

```

```

            p->liste->suiv = NULL;

```

```

            p->cour = p->liste;

```

```

            prec = NULL;

```

```

        }

```

```

        else

```

```

        {

```

```

            p->cour->suiv = malloc(sizeof(point));

```

```

            p->cour->suiv->x = n->p.x;

```

```

            p->cour->suiv->y = n->p.y;

```

```

            p->cour->suiv->suiv = NULL;

```

```

            prec = p->cour;

```

```

            p->cour = p->cour->suiv;

```

```

        }

```

```

        if ((n->dr != NULL) | (n->ga != NULL))

```

```

        {

```

```

            if (n->dr)

```

```

                plus_petit_parcours(n->dr, s, p);

```

```

            if (n->ga)

```

```

                plus_petit_parcours(n->ga, s, p);

```

```

            s -= n->val;

```

```

            free(p->cour);

```

```

            if (prec != NULL)

```

```

                prec->suiv = NULL;

```

```

            p->cour = prec;

```

```

        }

```

```

        else

```

```

        {

```

```

            if ((s < p->val) | (p->val == 0))

```

```

            {

```

```

        p->val = s;
        efface_liste(p->parcours);
        p->parcours = copie_liste(p->liste);
    }
    s -= n->val;
    free(p->cour);
    prec->suiv = NULL;
    p->cour = prec;
}
else
{
    if (n->dr)
        plus_petit_parcours(n->dr,s,p);
    if (n->ga)
        plus_petit_parcours(n->ga,s,p);
}
}

```

```

carre *suppr_carre(carre *car)
{
    carre *temp;

    while (car)
    {
        temp = car->suiv;
        free(car);
        car = temp;
    }
    return(NULL);
}

```

```

noeud *supprime_arbre(noeud *a)
{
    if (a->dr)
        a->dr = supprime_arbre(a->dr);

    if (a->ga)
        a->ga = supprime_arbre(a->ga);
    free(a);

    return(NULL);
}

```

```

void main()
{
    point *p;
    struct parc pa;

    creer_sortie();
    if (init())
    {
        arrive.x = 100; arrive.y = 100;
        pa.val = 0.0; pa.liste = NULL;
        pa.parcours = NULL;

        arbre = creer_arbre(NULL,arrive,arrive);
        plus_petit_parcours(arbre,0.0f,&pa);
        lcarre= suppr_carre(lcarre);
        arbre = supprime_arbre(arbre);
    }
}

```

```

        // inscrit le resultat dans le fichier de sortie
        p = pa.parcours;
        while(p)
        {
            fprintf(sortie, "%d %d\n", p->x, p->y);
            p = p->suiv;
        }
        fclose(sortie);
    }
}

```

C) IMAGE

Le principe est le suivant :

Le problème peut se résoudre de deux manières différentes, soit par un parcours de ligne, soit par une sommation des périmètres des images. Nous présenterons les deux solutions, une en Pascal et l'autre en C.

1) Parcours de ligne

L'idée d'un algorithme de parcours de ligne est que chaque ligne horizontale peut couper certaines images. L'intersection pouvant être facilement déterminée par comparaison des coordonnées Y.

Donc, en parcourant la ligne de gauche à droite, nous pouvons implémenter un compteur qui soit incrémenté chaque fois que l'on coupe l'arête gauche d'une image et décrémente chaque fois que l'on coupe l'arête droite d'une image. Le périmètre est augmenté à chaque fois que le compteur passe de 0 à une autre valeur ou lorsqu'il revient à 0. Seuls ces changements contribuent au calcul du périmètre, mais pour obtenir le résultat final, il faut aussi appliquer cette technique aux lignes verticales. Une possibilité dans ce cas est de permuter les coordonnées X et Y et de réappliquer un parcours de ligne horizontal.

2) Sommation de périmètres

Dans ce cas, le périmètre est obtenue en additionnant les périmètres de chaque image. Nous ne considérons pas que les images d'origine, obtenues en lisant le fichier d'entrée (IMAGE.IN). En effet, les intersections entre chaque paire d'images doivent être traitées comme une nouvelle image et le périmètre correspondant doit être déduit du périmètre global. Cette solution peut pour N images générer $2^N - 1$ images.

Complexité

Dans la première solution, le parcours de ligne, la complexité est fonction du nombre d'images en entrée et de la valeur des coordonnées (position des images), tandis que celle de la sommation de périmètres est exponentiellement dépendante du nombre d'images et du degré de recouvrement de celles-ci.

Programme Pascal (TP7)

```

program Image;
Uses Crt;
type
    pRect = ^rectangle;
    rectangle = record
        xmin, ymin, xmax, ymax: integer;
        suivant: pRect
    end;

    pArete = ^Arete;

    Arete = record
        {Arêtes verticales}
        sorte: (left, right);

```

```

    termine: boolean;
    x, ymin, ymax: integer;
    suivant: pArete
end;

var
  LRy, R: pRect;    { LRy = liste des rectangles }
  LEx: pArete;
  i, N, yminG, ymaxG, intRect, bordRect: integer;
  f, g: text;
  Perimetre: longint;

procedure miseajour (ymin, ymax: integer; var yminG, ymaxG: integer);
begin
  if ymin < yminG then yminG := ymin;
  if ymax > ymaxG then ymaxG := ymax;
end;

procedure InsereRectangle (var Ly, R: pRect);
var
  pointeur, precedent: pRect;
  moins: boolean;
begin
  if Ly = nil then
  begin
    R^.suivant := nil;
    Ly := R
  end
  else
  begin
    moins := true;
    pointeur := Ly;
    precedent := Ly;
    while (pointeur <> nil) and moins do
    begin
      if pointeur^.ymin >= R^.ymin then
        moins := false
      else
      begin
        precedent := pointeur;
        pointeur := pointeur^.suivant
      end
    end;
    if pointeur = Ly then Ly := R
    else precedent^.suivant := R;
    R^.suivant := pointeur
  end
end;

procedure ChoixAretes (y: integer; var LRy: pRect; var LEx: pArete);
var
  LRAux: pRect;
  E: pArete;
  plusgrand: boolean;

  procedure InsereArete (E: pArete; var LEx: pArete);
  var
    LEAux, precedent: pArete;
    moins: boolean;
  begin {Ordre croissant de x}
    if LEx = nil then

```

```

begin
    E^.suivant := nil;
    LEx := E
end
else
begin
    moins := true;
    LEAux := LEx;
    precedent := LEx;
    while (LEAux <> nil) and moins do
        if LEAux^.x >= E^.x then moins := false
        else
            begin
                precedent := LEAux;
                LEAux := LEAux^.suivant
            end;
        if LEAux = LEx then LEx := E
        else precedent^.suivant := E;
        E^.suivant := LEAux
    end
end;

begin {ChoixAretes}
    plusgrand := false;
    while (LRy <> nil) and (not plusgrand) do
        if LRy^.ymin = y then
            begin
                LRAux := LRy;
                new(E);
                E^.sorte := left;
                E^.termine := false;
                E^.x := LRy^.xmin;
                E^.ymin := LRy^.ymin;
                E^.ymax := LRy^.ymax;
                InsereArete(E, LEx);
                new(E);
                E^.sorte := right;
                E^.termine := false;
                E^.x := LRy^.xmax;
                E^.ymin := LRy^.ymin;
                E^.ymax := LRy^.ymax;
                InsereArete(E, LEx);
                LRy := LRy^.suivant;
                dispose(LRAux)
            end
        else plusgrand := true
    end;

procedure SupprimeAretes (y: integer; var LEx: pArete);
var
    aux, prev: pArete;
    found: boolean;
begin {Supprime les arêtes avec ymax=y-1; repère les Arêtes avec ymax=y}
    aux := LEx; {LEx<>nil }
    repeat
        found := aux^.ymax = y - 1;
        if found then
            begin
                aux := aux^.suivant;
                dispose(LEx);
                LEx := aux ;
            end
        end
    until not found;
end;

```

```

        end
until (aux = nil) or (not found);
prev := aux;
while aux <> nil do
    if aux^.ymax = y - 1 then
        if prev <> aux then
            begin
                prev^.suivant := aux^.suivant;
                dispose(aux);
                aux := prev^.suivant
            end
        else
            begin
                LEx := aux^.suivant;
                prev := LEx;
                dispose(aux);
                aux := LEx
            end
        end
    else
        begin
            aux^.termine := aux^.ymax = y;
            prev := aux;
            aux := aux^.suivant
        end
    end
end;

procedure ParcoursAretes (y: integer; Lx: pArete);
var
    intRect, oldIntRect, totalRect, oldTotalRect,
    lowBRect, oldLowBRect, subtr, oldSubtr: integer;
    prev: pArete;
    first: boolean;
begin
    intRect := 0;
    oldIntRect := 0;
    totalRect := 0;
    oldTotalRect := 0;
    lowBRect := 0;
    oldLowBRect := 0;
    subtr := 0;
    oldSubtr := 0;
    prev := Lx;
    first := true;
    repeat
        if Lx^.termine then
            if Lx^.sorte = left then
                subtr := subtr + 1
            else subtr := subtr - 1
            else
                begin
                    if Lx^.sorte = left then
                        begin
                            totalRect := totalRect + 1;
                            if Lx^.ymax > y + 1 then intRect := intRect + 1;
                            if Lx^.ymin = y then lowBRect := lowBRect + 1
                        end
                    else {right Arete}
                        begin
                            totalRect := totalRect - 1;
                            if Lx^.ymax > y + 1 then intRect := intRect - 1;
                            if Lx^.ymin = y then lowBRect := lowBRect - 1

```

```

        end;
        if (totalRect = 0) and (oldTotalRect = 1) then
            Perimetre := Perimetre + 1
        else if (totalRect = 1) and (oldTotalRect = 0) then
            if not first and (prev^.x = Lx^.x) and not
prev^.termine then
                Perimetre := Perimetre - 1
            else
                Perimetre := Perimetre + 1;
                if (oldIntRect = 0) and (oldTotalRect > 0) then
                    Perimetre := Perimetre + Lx^.x - prev^.x;
                    if (oldLowBRect > 0) and (oldTotalRect =
oldLowBRect) then
                        begin
                            Perimetre := Perimetre + Lx^.x - prev^.x;
                            if oldSubtR > 0 then
                                Perimetre := Perimetre - 2 * (Lx^.x -
prev^.x)
                            end
                        end;
                    end;
                    oldIntRect := intRect;
                    oldTotalRect := totalRect;
                    oldLowBRect := lowBRect;
                    oldSubtR := subtR;
                    first := false;
                    prev := Lx;
                    Lx := Lx^.suivant
                until Lx = nil;
            end;
        begin { PRINCIPAL }
            assign(f, 'IMAGE.IN');
            reset(f);
            Perimetre := 0;
            read(f, N);
            new(R);
            with R^ do
                begin
                    read(f, xmin, ymin, xmax, ymax);
                    yminG := ymin;
                    ymaxG := ymax;
                end;
            LRy := nil;
            InsereRectangle(LRy, R);
            for i := 2 to N do
                begin
                    new(R);
                    with R^ do
                        begin
                            read(f, xmin, ymin, xmax, ymax);
                            miseajour(ymin, ymax, yminG, ymaxG)
                        end;
                    InsereRectangle(LRy, R)
                end;
            close(f);
            LEx := nil;
            for i := yminG to ymaxG do
                begin
                    ChoixAretes(i, LRy, LEx);
                    if LEx <> nil then
                        begin

```

```

                SupprimeAretes(i, LEx);
            if LEx <> nil then ParcoursAretes(i, LEx);
        end
    end;
    writeln('Périmètre=', Perimetre : 1);
    assign(g, 'IMAGE.OUT');
    rewrite(g);
    writeln(g, Périmètre:1);
    close(g)
end.

```

Programme C (GCC)

```

/* Programme IMAGE */

#include <stdio.h>

#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) > (b) ? (b) : (a))

typedef struct{
                int sign, x, y, X, Y; /* niveau, Coin Inf Gauche & Sup
Droit */
                } pict;

#define MAX_PICT 10000

void intersect (pict p1, pict p2, pict *p3)
{
    p3->x = max(p1.x, p2.x); p3->y = max(p1.y, p2.y);
    p3->X = min(p1.X, p2.X); p3->Y = min(p1.Y, p2.Y);
    p3->sign = -p1.sign * p2.sign;
}

int main ()
{
    long perim = 0;
    long npf, npt = 0; /* nombre d'images dans le fichier et nombre total
*/
    FILE *f;
    pict p[MAX_PICT];
    int i, j, k;

    if ((f = fopen("PICTURE.IN", "r")) == NULL)
    {
        printf("Fichier vide!\n");
        return 0;
    }
    fscanf(f, "%d\n", &npf);
    for(i = 0; i < npf ; i++)
    {
        k = npt++;
        fscanf(f, "%d %d %d %d\n", &p[k].x, &p[k].y, &p[k].X, &p[k].Y);
        p[k].sign = 1;
        for (j = 0; j < k; j++)
        {

```

```

        intersect(p[j], p[k], &p[npt]);
        if ((p[npt].X > p[npt].x && p[npt].Y >= p[npt].y) ||
            (p[npt].X == p[npt].x && p[npt].Y > p[npt].y)) /* p[npt]
existe */
            if (p[npt].x == p[k].x && p[npt].X == p[k].X &&
                p[npt].y == p[k].y && p[npt].Y == p[k].Y)
            { /* images identiques */
                npt = k;
                break;
            }
            else npt++;
    }
}
fclose(f);
for (i = 0; i < npt; i++)
    perim += p[i].sign * 2 * (p[i].X - p[i].x + p[i].Y - p[i].y);
printf ("Perimeter=%ld\n", perim);
f=fopen("PICTURE.OUT", "w");
fprintf(f, "%ld\n", perim);
fclose(f);
}

```