Permutations : les trois méthodes récursives

Première méthode : la plus compréhensible. C'est celle que nous utilisons dans de nombreux problèmes (cf. livre *Combien ? tome 1, chapitre* 4, *page 78*, ou en anglais *Mathematics for computer science, chapter 4, page 66*)

```
#define N 11
int compteur, pred[N];

int main()
{   int start;
    for(start=0; start<N; start++) arbre(start,0);
    printf("%d",compteur);    return 0;
}
void arbre (int i, int etage)
{  int j,k;
    if (etage==N-1)
       { compteur++;
         /* printf("\n%d : ",compteur);   si l'on veut afficher les permutations
            for (j=1;j<N;j++) printf("%d ", pred[j]);                    printf("%d ", i);
          */
       }
    else
       { for(k=0;k<N;k++) if (k!=i)
          if (appartientlistepred(k,etage)==0)   { pred[etage+1]=i;    arbre(k,etage+1); }
       }
}
int appartientlistepred(int j,int etage)
  {   int et;
      for(et=1;et<=etage;et++) if (pred[et]==j) return 1;
      return 0;
  }
```

Deuxième méthode : la meilleure, mais elle est plus abstraite que la précédente, et nécessite une bonne connaissance de la récursivité.

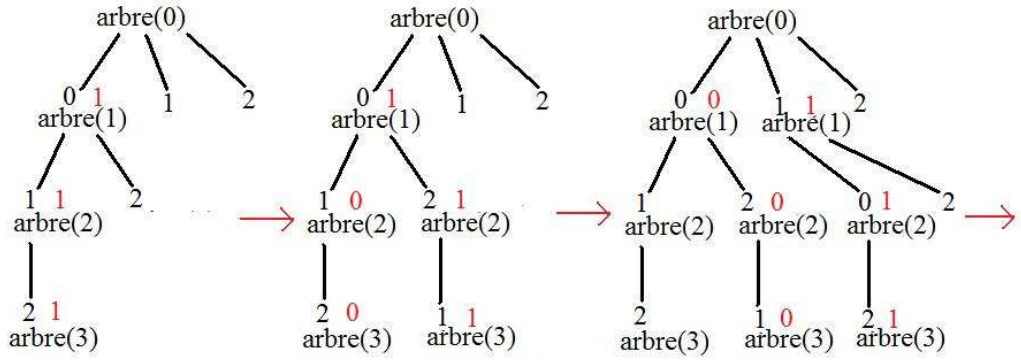
```
#define N 10
int finished[N],p[N],compteur;

int main()
   {   arbres(0);   return 0; }

void arbres (int etage)
{  int i,j;
    if (etage==N)
       { compteur++;
         /*printf("\n%d : ",compteur);
         for (j=0;j<N;j++) printf("%d ", p[j]);
         */
       }
```

```
    else
      { for(i=0;i<N;i++)
        if (finished[i]==0)
          { finished[i]=1; p[etage]=i;
            arbres(etage+1);
            finished[i]=0;
          }
      }
}
```

Dans le dessin ci-dessous, on voit comment évolue l'arborescence dans le cas où $N = 3$, avec la variable *finished* (en rouge) qui se met à 1 en descente, et à 0 lors de la remontée.



Troisième méthode : les liens dansants. On reprend ici l'algorithme de la couverture exacte (*voir travaux complémentaires, le problème de l'exact cover*), avec un ensemble à $NB = 2N$ éléments, où les N premiers correspondent à un nombre de 0 à $N - 1$, et où les N derniers correspondent à la position du nombre considéré. Les $NBSE$ parties, au nombre de N^2, sont obtenues ainsi :

```
for(i=0;i<NBSE;i++) {   se[i][0]=i%N; se[i][1]=N+i/N;  }
```

Par exemple, pour $N = 3$, les parties sont:
{0 3}, {1 3}, {2 3}, {0 4}, {1 4}, {2 4}, {0 5}, {1 5}, {2 5}.
Ainsi, la partie {0 3}indique que le nombre 0 est en première position, la partie {2 4] indique que le nombre 2 est en deuxième position, etc. Une couverture exacte est obtenue en prenant trois parties, comme par exemple {0 3}, {1 5}, {2 4}, celle-ci correspondant à une permutation. Le nombre de couvertures exactes est aussi le nombre des permutations.

```
#define NB (2*N)
#define NBSE (N*N)
int se[NBSE][NB];
int resultat[100],n,compteur;
struct cell
  { int l; int c;
    struct cell * d; struct cell * g; struct cell * b; struct cell * h;
  };
struct cell * racine, * newcell, *oldcell,* debutligne[NBSE],
    * debutcolonne[NB],*oldcelcol[NB+1];
```

```
int main()
{   entreedesparties();
    matrice();
    chercher();
    printf(" %d",compteur);
    return 0;
}

void entreedesparties(void)
{  int i;
   for(i=0;i<NBSE;i++) {   se[i][0]=i%N; se[i][1]=N+i/N;  }
}

void matrice(void)
{   int i,j,colonne;
    racine=(struct cell *) malloc(sizeof(struct cell));
    racine->l=-1, racine->c=-1; racine->b=racine; racine->h= racine;
    racine->d=racine; racine->g= racine;
    /** bordure gauche verticale */
    oldcell=racine;
    for(i=0;i<NBSE; i++)
    {
       debutligne[i]=(struct cell *) malloc(sizeof(struct cell));
       debutligne[i]->l=i; debutligne[i]->c=-1;
       oldcell->b=debutligne[i];
       racine->h=debutligne[i];
       debutligne[i]->b=racine; debutligne[i]->h=oldcell;
       oldcell=debutligne[i];
    }
    /** bordure haute horizontale */
    oldcell= racine;
    for(j=0;j<NB; j++)
    {
       debutcolonne[j]=(struct cell *) malloc(sizeof(struct cell));
       debutcolonne[j]->l=-1; debutcolonne[j]->c=j;
       oldcell->d=debutcolonne[j];
       racine->g=debutcolonne[j];
       debutcolonne[j]->d=racine; debutcolonne[j]->g=oldcell;
       oldcell=debutcolonne[j];
    }
    /** construction des lignes comme listes chaînées */
    for(j=0;j<NB;j++) oldcelcol[j]=debutcolonne[j];

    for(i=0;i<NBSE;i++)
    { oldcell=debutligne[i];
      for(j=0;j<2;j++)
       { newcell=(struct cell *) malloc(sizeof(struct cell));
         colonne=se[i][j];
         newcell->l=i; newcell->c=colonne;
        newcell->d=debutligne[i]; newcell->g=oldcell;
        oldcell->d=newcell;   debutligne[i]->g=newcell;
        oldcell=newcell;
        newcell->h=oldcelcol[colonne]; newcell->b=debutcolonne[colonne];
        oldcelcol[colonne]->b=newcell;
        debutcolonne[colonne]->h=newcell;
```

```
          oldcelcol[colonne]=newcell;
        }
    }
/** supprimer les butoirs de lignes horizontalement */
for(i=0;i<NBSE;i++)  {(debutligne[i]->g)->d=debutligne[i]->d;
                      (debutligne[i]->d)->g=debutligne[i]->g;
                     }
}
/* pour vérifier que la matrice est bien enregistrée */
void affichercolonne(int j)
{  struct cell * ptr;
   printf("\n%d: ",j);
   ptr=debutcolonne[j]->b;
   do {
        printf("%d ",ptr->l);
        ptr=ptr->b;
       }
   while (ptr!=debutcolonne[j]);
}
void afficherlescolonnes(void)
{  struct cell * ptr;
   ptr=racine->d;
   while(ptr!=racine)
       {
         affichercolonne(ptr->c);
         ptr=ptr->d;
       }
}
void affichermatrice(void)
{  struct cell *ptrc, *ptrl;
   ptrc=racine->b;
   while(ptrc!=racine)
     { printf("\n%d:",ptrc->l);
       ptrl=ptrc->d;
       do
         {
           printf("%d ",ptrl->c);
           ptrl=ptrl->d;
         }
         while (ptrl!=ptrc->d);
       ptrc=ptrc->b;
     }

}
/* et maintenant les liens dansants */
void  cover(int col)
{  struct cell * ptrl, *ptrc;
   (debutcolonne[col]->g)->d=debutcolonne[col]->d;
   (debutcolonne[col]->d)->g=debutcolonne[col]->g;
   ptrc=debutcolonne[col]->b;
   while(ptrc!=debutcolonne[col])
     {
       ptrl=ptrc->d;
       while(ptrl!=ptrc)
         { (ptrl->h)->b=ptrl->b;  (ptrl->b)->h=ptrl->h;
```

```
            ptrl=ptrl->d;
          }
        ptrc=ptrc->b;
      }
}

void  uncover(int col)
{  struct cell * ptrl, *ptrc;

   ptrc=debutcolonne[col]->h;
   while(ptrc!=debutcolonne[col])
     {
       ptrl=ptrc->g;
       while(ptrl!=ptrc)
         { (ptrl->h)->b=ptrl;  (ptrl->b)->h=ptrl;
           ptrl=ptrl->g;
         }
       ptrc=ptrc->h;
     }
   (debutcolonne[col]->g)->d=debutcolonne[col];
   (debutcolonne[col]->d)->g=debutcolonne[col];
}

void chercher(void)
{    int colonne,i;
     struct cell * ptrc, *ptrl;

   if (racine->d==racine && racine->g==racine)
     { compteur++;
     /**printf(" \nsolution %d:",compteur);
       for(i=0;i<n;i++) printf(" %d ",resultat[i]);
     */
     }
   else
     {
       colonne=(racine->d)->c;
       cover(colonne);
       ptrc=debutcolonne[colonne]->b;
       while(ptrc!=debutcolonne[colonne])
         {
           resultat[n++]=ptrc->l;      ptrl=ptrc->d;
           while(ptrl!=ptrc)    {cover(ptrl->c);   ptrl=ptrl->d; }
           chercher();
           ptrl=ptrc->g;
           while(ptrl!=ptrc)   { uncover(ptrl->c);     ptrl=ptrl->g;      }
           ptrc=ptrc->b;
           resultat[--n]=0;
         }
       uncover(colonne);
     }
}
```