Jeu de taquin :
résolution automatique sur ordinateur

Sous sa forme la plus générale, le jeu de taquin est constitué d'un rectangle rempli par des blocs carrés accolés, chacun portant un numéro, avec un bloc laissé vide. Un des voisins de cette case carrée vide peut coulisser pour prendre cette place vide, la case vide prend alors la place du voisin. Tout se passe comme si la case vide se déplaçait à chaque coup avec une modification progressive de l'ordre des cases numérotées. En partant d'une configuration où les cases ont leurs numéros dans le désordre, avec la case vide située en bas à droite, il s'agit d'effectuer des déplacements de la case vide de façon à retrouver à la fin l'ordre naturel, comme sur la figure ci-dessus dans le cas d'un carré 4×4.

1. Remarques préliminaires pour la programmation

En prenant un rectangle avec *N* cases carrées en ligne et *P* cases en colonne, nous allons numéroter les *NP* – 1 cases de 0 à *NP* – 2, et la case vide portera le numéro masqué *NP* – 1. Une configuration quelconque des cases numérotées est une certaine permutation des numéros. Celle-ci sera enregistrée dans un tableau *a*[*NP*], les indices étant les positions dans l'ordre naturel.

Dans l'exemple de la *figure 1*, avec ***N*** = 4 et *P* = 2 comme dimensions du rectangle, le tableau *a*[] est ainsi rempli :

a[0] = 4, *a*[1] = 6, *a*[2] = 2, *a*[3] = 3, *a*[4] = 0, *a*[5] = 5, *a*[6] = 1, et pour la case vide *a*[7] = 7.

0	1	2	3
4	5	6	7

4	6	2	3
0	5	1	

Figure 1 : *A gauche*, les positions numérotées des cases, de gauche à droite et de haut en bas, *à droite* une configuration du jeu dans un certain désordre.

La connaissance du tableau *a*[] permet le dessin de la configuration correspondante, grâce à la fonction *dessin*() :

```
void dessin(void)
{ int i;
  for(i=0;i<=N;i++) line(xorig+pas*i,yorig,xorig+pas*i,yorig+P*pas,noir);      /* dessin de la grille des
  for(i=0;i<=P;i++) line(xorig,yorig+pas*i,xorig+N*pas,yorig+pas*i,noir);         carrés */
  for(i=0;i<NP;i++)
    { x[i]=i%N; y[i]=i/N; /* coordonnées du coin en haut à gauche du carré en position i*/
      if (a[i]!=NP-1)  /* on ne numérote pas la case vide */
      { sprintf( chiffre,"%d",a[i]);  /* dessin du numéro porté par chaque bloc carré */
        texte=TTF_RenderText_Solid(police,chiffre,couleurnoire);
        position.x=pas/5+xorig+pas*x[i]; position.y=pas/6+yorig+pas*y[i];
        SDL_BlitSurface(texte,NULL,screen,&position);
```

```
        }
      }
}
```

Par la même occasion, on peut avoir intérêt à utiliser un tableau des positions, *pos*[*NP*], qui donne la position de chaque numéro. Dans l'exemple précédent, on a *pos*[0] = 4, *pos*[1] = 6, etc., et l'on note en particulier *poscv* la position de la case vide, *poscv* n'est autre que *pos*[*NP* – 1]. Remarquons que le tableau *pos*[] correspond à la permutation inverse de *a*[].

Chaque mouvement de la case vide va provoquer des modifications dans ce tableau *a*[] et aussi dans le tableau *pos*[]. La case vide peut se déplacer dans quatre directions : monter ou descendre d'un cran, aller à droite ou à gauche, sauf si elle est en bordure du rectangle, où ses déplacements sont plus limités. Rappelons que la position de la case vide est enregistrée dans la variable (globale) *poscv*.

Supposons que la case vide monte d'un cran, dans le contexte de la *figure 2*, avec un rectangle où *N* = 5 et *P* = 4. Sa position est *poscv* = 13, et elle va passer en position 13 – *N* = 8, tandis que la case portant le numéro 18 va passer en position 13. Ainsi dans *a*[13] on met 18 (soit *a*[*poscv* – *N*]), et dans *pos*[18] on met 13 (soit *poscv*). Puis on déplace la case vide : *poscv* = 8, et *a*[8] = 19 (soit le numéro associé à la case vide *NP* – 1), avec *pos*[19] = *poscv*. On en déduit la fonction qui fait monter d'un cran la case vide :

13	15	3	1	5
14	4	7	18	2
8	16	12		10
6	0	9	17	11

Figure 2 : Une configuration, à partir de laquelle la case vide va monter pour s'échanger avec la case 18

```
void monteecv(void)
{  a[poscv]=a[poscv-N]; pos[a[poscv]]=poscv;
   poscv=poscv-N;a[poscv]=NP-1; pos[NP-1]=poscv;
}
```

On fait de même pour les déplacements dans les trois autres directions :

```
 void gauchecv(void)
 { a[poscv]=a[poscv-1]; pos[a[poscv]]=poscv;
   poscv=poscv-1;a[poscv]=NP-1; pos[NP-1]=poscv;
 }
 void descentecv(void)
{  a[poscv]=a[poscv+N]; pos[a[poscv]]=poscv;
   poscv=poscv+N;a[poscv]=NP-1; pos[NP-1]=poscv;
}
void droitecv(void)
 { a[poscv]=a[poscv+1]; pos[a[poscv]]=poscv;
   poscv=poscv+1;a[poscv]=NP-1; pos[NP-1]=poscv;
 }
```

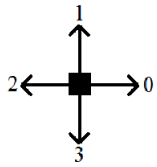
Nous allons maintenant donner deux méthodes de résolution du jeu de taquin. La première méthode est purement informatique, tandis que la deuxième fait faire à l'ordinateur ce que l'on a tendance à faire à la main. Dans une dernière partie, nous donnerons une troisième méthode, plus proche de la théorie du jeu de taquin.

2. Première méthode de résolution automatique du jeu

Nous allons partir de la configuration initiale dans l'ordre naturel, avec la case vide en bas à droite. Puis nous allons introduire du désordre en provoquant un grand nombre de mouvements au hasard de la case vide, que nous enregistrons par la même occasion. On fait aussi en sorte que la case vide se retrouve en bas à droite. Au terme de cette première étape, on aura une configuration désordonnée du taquin, mais en sachant de quels mouvements elle provient. A partir de là, comment revenir à l'ordre naturel ? Il suffit de faire les mouvements inverses de la case vide, puisque nous les connaissons. Au terme de cette deuxième étape de remise en ordre, on a résolu le problème.

Première étape : de l'ordre au désordre

On part de l'ordre naturel, ce qui revient à mettre dans le tableau *a*[] les numéros successifs de 0 à *NP* – 1. Précisons que dans cette méthode, seul le tableau *a*[] devra être géré au fil des déplacements de la case vide, et que le tableau inverse *pos*[] ne sert à rien.[1] Puis on va provoquer un grand nombre de mouvements aléatoires de la case vide, via la répétition d'une fonction *mouvcv*(). A chaque fois, le case vide va se déplacer au hasard vers une des cases voisines. Mais ces cases voisines peuvent être au nombre de 4, mais aussi 3 ou 2 lorsque l'on est sur la bordure du rectangle. D'où la mise en place d'une fonction *grille*() qui détermine pour chaque case quelles sont ses voisines ainsi que leur nombre.



Pour la case en position *i*, les positions de ses voisins sont enregistrées dans le tableau *v*[*i*][*k*] et le nombre de voisins dans *nbv*[*i*]. Comme un voisin est situé soit à droite, soit au-dessus, soit à gauche, soit au-dessous, on enregistre aussi dans quelle direction il est placé, avec les nombres 0, 1, 2, ou 3 placés dans un tableau *d*[*i*][*k*].

```
void grille(void)
 { int i,k;
   for(i=0;i<NP;i++)
     { k=0;
       if ((i+1)%N!=0) { v[i][k]=i+1; d[i][k]=0;k++;} /* il existe un voisin à droite */
       if (i/N!=0) { v[i][k]=i-N; d[i][k]=1;k++;}  /* il existe un voisin au-dessus */
       if (i%N!=0) { v[i][k]=i-1; d[i][k]=2;k++;}  /* il existe un voisin à gauche */
       if (i<NP-N) { v[i][k]=i+N; d[i][k]=3;k++;} /* il existe un voisin au-dessous */
       nbv[i]=k;
     }
}
```

Passons à la fonction *mouvcv*(). La case vide étant en position *poscv* à une étape numérotée *compteur*, il s'agit de choisir au hasard une des *nbv*[*poscv*] cases voisines, tout en évitant de revenir là où se trouvait la case vide à l'étape précédente, afin de ne pas faire des allers-retours inutiles. D'où la gestion d'une variable *oldposcv* qui garde en réserve la position précédente de la case vide. Puis selon la position de la case voisine, que l'on avait enregistrée dans le tableau *d*[][], on provoque le mouvement correspondant de la case vide. Par la même occasion on mémorise cette direction de déplacement dans un tableau *b*[], soit *b*[*compteur*] à l'étape *compteur* du mouvement. C'est grâce à ce tableau *b*[] que l'on peut suivre les mouvements successifs de la case vide de l'ordre au désordre, de façon à pouvoir plus tard les effectuer en sens inverse, pour revenir à l'ordre.

```
void mouvcv(void)
{ int h;
   do  {h=rand()%nbv[poscv];}  while(v[poscv][h]==oldposcv); /* choix au hasard d'une case voisine */
   oldposcv=poscv;
   b[compteur++]=d[poscv][h];  /* mémorisation du mouvement dans le tableau b[] */
   if (d[poscv][h]==0) droitecv();   /* suivant la direction prise, on fait bouger la case vide */
```

[1] Cela permet de simplifier les quatre fonctions de déplacements de la case vide, soit *monteecv*(), *gauchecv*(), etc., où les références au tableau *pos*[] peuvent être supprimées.

```
    else if (d[poscv][h]==1) monteecv();
    else if (d[poscv][h]==2) gauchecv();
    else if (d[poscv][h]==3) descentecv();
}
```

Au terme de l'action répétée de cette fonction *mouvcv*(), le désordre est obtenu, avec la case vide quelque part dans le rectangle. Il ne reste plus qu'à la déplacer pour la replacer en bas à droite, en provoquant un déplacement vertical *diffvert* puis un déplacement horizontal *diffhor*. Ce qui donne ce début de programme principal permettant d'aller de l'ordre au désordre.

```
for(i=0;i<NP;i++) a[i]=i;  /* rectangle avec l'ordre naturel pour les blocs carrés */
grille();dessin();SDL_Flip(screen); pause();
oldposcv= -1; poscv=NP-1; /* conditions initiales pour la case vide, qui a deux possibilités de mouvements,
                             la valeur initiale de oldpcv ne provoquant aucun empêchement */
for(i=0;i<30000;i++) mouvcv();  /* déplacements répétés de la case vide */
diffvert=P-1-poscv/N; /* différence verticale entre la position actuelle et la position finale de la case vide */
for(i=0;i<diffvert;i++) {descentecv(); b[compteur++]=3;} /* mémorisation des mouvememts dans b[] */
diffhor=N-1-poscv%N;  /* de même horizontalement */
for(i=0;i<diffhor;i++) {droitecv(); b[compteur++]=0;}
SDL_FillRect(screen,0,blanc);dessin();SDL_Flip(screen);pause();
```

Un résultat de ce morceau de programme est donné sur la *figure 3*. Précisons que les déplacements successifs, enregistrés dans le tableau *b*[], comportent autant de 0 que de 2, et de 1 que de 3, car la case vide revient à la fin là où elle était au départ.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	

13	2	15	12	7
9	17	1	10	0
18	8	5	14	6
3	16	4	11	

Figure 3 : Passage de l'ordre (*à gauche*) au désordre (*à droite*)

Deuxième étape : du désordre à l'ordre

Les déplacements de la case vide ont été enregistrés dans le *tableau b*[], et ils sont au nombre de *compteur*. Pour revenir à l'ordre, il suffit de les faire du dernier au premier, et en sens inverse, une montée étant par exemple transformée en descente. On aboutit à cette fin du programme principal :

```
for(i=compteur-1;i>=0;i--)
  { if (b[i]==0) gauchecv();
    else if (b[i]==1) descentecv();
    else if (b[i]==2) droitecv();
    else if (b[i]==3) monteecv();
  }
SDL_FillRect(screen,0,blanc);dessin();SDL_Flip(screen);pause();
```

3. Variante : reconstitution d'un dessin

Prenons une image inscrite dans un cadre rectangulaire, et découpons-la suivant une grille de blocs carrés. On se retrouve alors dans le contexte du jeu de taquin, où les blocs carrés numérotés sont remplacés par des morceaux d'images. Il suffit d'appliquer l'algorithme précédent, légèrement réaménagé, pour d'abord construire un assemblage désordonné des morceaux d'image, puis par déplacement de la case vide (ici un bloc carré de l'image, celui qui est situé en bas à droite au début de l'opération) reconstituer l'image initiale.

Un résultat est donné sur la *figure 4*. Précisons que l'image initiale, de 360 sur 360 pixels, a été découpée en 12 ×12 carrés de longueur L = 360 / 12 = 30. On a effectué une dizaine de milliers de déplacements de la case vide, pour la mise en désordre, ainsi que pour la remise en ordre. Précisons que par cette méthode l'image reconstituée ne réapparaît pas avant les derniers mouvements de la case vide.

Figure 4 : *A gauche*, les morceaux de l'image dans le désordre, *à droite*, après de multiples déplacements de la case vide, reconstitution finale de l'image, ici une peinture de Diego Rivera

4. Deuxième méthode de résolution

Maintenant l'ordinateur va agir comme le ferait un joueur humain. A partir d'une permutation quelconque des cases numérotées, la remise en ordre se fait progressivement. D'abord on déplace la case 0 pour la remettre à sa place en haut à gauche. Puis on fait de même pour la case 1, et ainsi de suite jusqu'à la dernière case. Par cette méthode, la remise en ordre se fait ligne après ligne à partir du haut. Si l'on utilise une image découpée en carrés au lieu de carrés numérotés, l'image va réapparaître progressivement du haut vers le bas, à la différence de la méthode du *paragraphe 3*.

Si le principe de la méthode est simple, sa réalisation pratique est plus complexe, car lorsque l'on veut remettre à sa bonne place un bloc carré, il convient que les déplacements de la case vide ne perturbent pas la position des cases déjà remises à leur place.

On voit sur la *figure 5* une situation où toutes les cases situées dans la zone délimitée par un trait rouge sont à leur place définitive. A l'étape suivante, c'est la case 28 qui viendra remplacer la case 44 grâce aux déplacements de la case vide.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	44	33	37	38
48	45	32		29	51	39	35
42	46	41	50	40	53	28	43
52	49	47	31	36	30	54	34

Figure 5 : Une situation intermédiaire du jeu, avec les blocs remis à leur place définitive de haut en bas et de gauche à droite dans la zone entourée de rouge. C'est ensuite la case 28 qui viendra prendre la place de la case 44.

La programmation se fait en plusieurs étapes. Commençons par la mise en place d'une configuration désordonnée des blocs carrés.

4.1. La permutation initiale

On sait, grace à la théorie, que le passage du désordre à l'ordre dans le jeu de taquin nécessite d'avoir au départ une permutation paire.[2] Sachant que le fait de procéder à un échange entre deux nombres de la permutation (à l'exclusion de la case vide) provoque toujours un changement de parité de la permutation, il suffit de partir de l'ordre naturel, qui est une permutation paire, et de faire un grand nombre d'échanges, ce nombre étant pair. On est ainsi assuré d'avoir à la fin une permutation paire, obtenue aléatoirement. D'où la fonction correspondante :

```
void permutation(void)
{ int i,j,aux,echange;
   for(i=0;i<NP-1;i++) a[i]=i;  /* ordre naturel */
   for(echange=0;echange<10000;echange++)
     { i=rand()%(NP-1);  do {j=rand()%(NP-1); } while (j==i);
       aux=a[i];a[i]=a[j];a[j]=aux;   /* l'échange */
     }
   a[NP-1]=NP-1;  /* la case vide */
   for(i=0;i<NP-1;i++) pos[a[i]]=i; /* après le tableau a[], construction du tableau pos[] */
   poscv=NP-1;  /* position de la case vide */
}
```

4.2. Comment déplacer une case ?

Le déplacement d'un bloc carré numéroté *i* se fait en deux temps. D'abord on envoie la case vide se placer dans le voisinage de la case numéro *i*, puis par des mouvements de la case vide, on va faire progresser le bloc *i* jusqu'à sa position finale *i*.

4.2.1. Mouvement de la case vide vers la case à déplacer

Il s'agit de déplacer la case vide jusqu'à ce qu'elle vienne se coller à côté du bloc carré que l'on veut ensuite changer de place. Nous avons decidé de placer la case vide juste à droite ou à gauche de la case concernée portant le numéro *i*.

Pour ce faire, on détermine d'abord la distance verticale *diffverticale* ainsi que la distance horizontale *diffhorizontale* qui sépare la case vide de la case à déplacer. Dans le programme, *diffverticale* est positive quand la case vide monte, et *diffhorizontale* est positive quand la case vide va vers la gauche. On distingue alors huit cas de figure illustrés sur la *figure 6*.

Remarquons que dans un cas, avec *diffverticale* > 0 et *diffhorizontale* < 0, il convient de commencer par le déplacement horizontal et de finir par le vertical, afin de ne pas risquer de traverser la zone des cases déjà bien rangées. On doit faire de même dans un autre cas, celui où *diffverticale* < 0 et *diffhorizontale* < 0, mais cela ne s'expliquera que plus tard, dans le *paragraphe 4.2.4.* avec la *figure 19*.

---

[2] Une permutation est paire si le nombre d'inversions est pair. Une inversion se produit lorsqu'en prenant deux éléments *a*[*i*] et *a*[*j*], avec *i* < *j*, de la permutation, on a *a*[*i*] > *a*[*j*]. Il suffit alors de compter le nombre d'inversions et de voir s'il est pair ou impair. Le petit programme suivant compte le nombre d'inversions (noté *nbinv* et mis à 0 au départ) :

```
for(i=0;i<NP-2;i++) for(j=i+1;j<NP-1;j++)
if(a[i]>a[j]) nbinv++;
```

Remarquons aussi que dans trois cas, le mouvement se termine par un crochet soit vers la droite soit vers la gauche. On privilégiera le crochet à droite. On ne prendra le crochet à gauche que dans le cas où la case à déplacer se trouve dans la dernière colonne du plateau rectangulaire, car l'on n'a pas d'autre choix. Le fait que la case vide soit à droite ou à gauche de la case à déplacer est enregistré dans une variable *flag* mise à 1 si c'est à droite et – 1 si c'est à gauche. On en déduit le programme de la fonction *casevidecolle* (*i*) où *i* est le numéro de la case à déplacer (et non pas sa position qui est *pos*[*i*]).

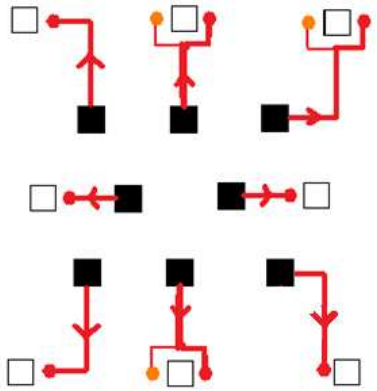


Figure 6 : Les déplacements de la case vide (dessinée en noir) pour aller se coller à droite ou à gauche de la case à déplacer (en blanc). En haut *diffvert* est positive, en dessous *diffvert* est nulle, en bas *diffvert* est négative

```
void casevidecolle(int i)
 { int diffverticale,diffhorizontale,j,xfinale,yfinale,q;
    xcv=poscv%N; ycv=poscv/N;
    xfinale=pos[i]%N; yfinale=pos[i]/N;
   diffverticale=ycv-yfinale;
   diffhorizontale=xcv-xfinale;
   if (diffverticale>=0 && diffhorizontale>0
     { for(j = 0;j<diffverticale;j++)
         { monteecv(); dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
         }
       for(j = 0;j<diffhorizontale-1;j++)
         { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc);
         }
       flag=1;
     }
   else if (diffverticale>0 && diffhorizontale==0)
     { for(j = 0;j<diffverticale-1;j++)
         { monteecv();
           dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
         }
       if (pos[i]%N<N-1)
         { droitecv();dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
           monteecv(); dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
           flag=1;
         }
        else if (pos[i]%N==N-1)
          { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
            monteecv(); dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
            flag=-1;
          }
     }
   else if (diffverticale>0 && diffhorizontale<0)
```

```
  {  for(j = 0;j<-diffhorizontale;j++)
          { droitecv();dessin();SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc);
          }
     for(j = 0;j<diffverticale-1;j++)
          { monteecv(); dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
          }
     if (pos[i]%N<N-1)
         { droitecv(); dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
           monteecv();dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
          flag=1;
         }
     else
        {gauchecv(); dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
         monteecv();dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
         flag=-1;
        }
  }
else if (diffverticale==0 && diffhorizontale>0)
  { for(j = 0;j<diffhorizontale-1;j++)
       { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc);
       }
    flag=1;
  }
else if (diffverticale==0 && diffhorizontale<0)
  { for(j = 0;j<-diffhorizontale-1;j++)
       { droitecv();dessin();SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc);
       }
    flag=-1;
  }
else if (diffverticale<0  && diffhorizontale>0)
  { for(j = 0;j<-diffverticale;j++)
       { descentecv();
         dessin();  SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc);
       }
    for(j = 0;j<diffhorizontale-1;j++)
       { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc);
       }
    flag=1;
   }
 else if (diffverticale<0 && diffhorizontale==0)
  { for(j = 0;j<-diffverticale-1;j++)
      { descentecv();
        dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
      }
    if (pos[i]%N<N-1)
      { droitecv();dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
        descentecv(); dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
        flag=1;
      }
     else if (pos[i]%N==N-1)
      { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
        descentecv(); dessin();SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc);
        flag=-1;
      }
  }
 else if (diffverticale<0 && diffhorizontale<0)
  {
    for(j = 0;j<-diffhorizontale-1;j++)
      { droitecv();dessin();SDL_Flip(screen); SDL_Delay(50);;SDL_FillRect(screen,0,blanc); }
    for(j = 0;j<-diffverticale;j++)
```

```
        { descentecv(); dessin();  SDL_Flip(screen); SDL_Delay(50);SDL_FillRect(screen,0,blanc); }
      flag=-1;        }
    }
  }
```

4.2.2. Déplacement d'une case vers sa position finale

Rappelons qu'en l'état actuel, la case vide se trouve soit juste à droite soit juste à gauche de la case numérotée *i*, celle que nous voulons déplacer jusqu'à sa position finale *ifinal* qui est aussi *i*, mais on verra qu'avant de parvenir à la position définitive *i* il conviendra parfois de passer par une position intermédiaire, d'où la distinction entre *i* et *ifinal*. On appellera *mouvement*(*i*, *ifinal*) la fonction qui fera ce déplacement de la case numéro *i* vers sa position finale *ifinal* (qui est soit *i* soit une position intermédiaire).

Dans un premier temps, on définit les moyens qui permettent à la case concernée de se déplacer d'un cran dans une des quatre directions possibles. Pour une montée ou une descente, avec la case vide située soit à droite soit à gauche, on effectue un mouvement tournant de la case vide, de façon que la case vide se retrouve du meme côté après son déplacement d'un cran. Sur la *figure 7* est indiquée la montée d'un cran d'une case lorsque la case vide est à sa gauche, et l'on en déduit la function *montéepargauche*(*d*) lorsque l'on veut monter de *d* crans.

Figure 7 : Exemple de la montée d'une case (en gris) avec la case vide (en noir) située à sa gauche. Le mouvement tournant de la case vide est indiqué en rouge.

On en déduit les programmes :

```
void monteepardroite(int d)
{ int j;
   for(j=0;j<d;j++)  { monteecv(); gauchecv(); descentecv(); droitecv();  monteecv();}
}
void monteepargauche(int d)
 {  int j;
    for(j=0;j<d;j++)  { monteecv(); droitecv(); descentecv(); gauchecv(); monteecv(); }
 }
void descentepardroite(int d)
  { int j;
    for(j=0;j<d;j++)  { descentecv(); gauchecv();  monteecv(); droitecv(); descentecv(); }
  }
void descentepargauche(int d)
  { int j;
    for(j=0;j<d;j++) { descentecv();  droitecv(); monteecv(); gauchecv(); descentecv();  }
  }
```

On fait de même avec les déplacements horizontaux, mais le mouvement tournant peut se faire soit par dessous soit par dessus. Sur la *figure 8* on voit comment faire bouger la case d'un cran à gauche avec un mouvement tournant qui se fait au-dessous, ce qui permet de construire la fonction *horizontalgauchebas*(*d*) lorsque le déplacement se fait sur *d* crans.

Remarquons que pour un déplacement à gauche, la case vide est toujours supposée être située à droite au début, et elle le reste à la fin, tandis que pour un déplacement à droite, elle est supposée située à gauche et le reste.

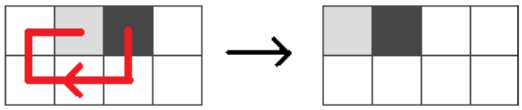


Figure 8 : Déplacement d'un cran à gauche de la case (en gris) grâce au mouvement tournant indiqué en rouge, la case vide (en noir) restant à droite.

On en déduit les fonctions correspondantes :

```
void horizontalgauchebas(int d) /* la case vide s'enroule par en dessous */
 {  int j;
    for(j=0;j<d;j++)  { descentecv(); gauchecv(); gauchecv(); monteecv(); droitecv();  }
 }
void horizontalgauchehaut(int d) /* la case vide s'enroule en passant par dessus */
 {  int j;
    for(j=0;j<d;j++) { monteecv(); gauchecv(); gauchecv(); descentecv(); droitecv(); }
}
 void horizontaldroitebas(int d)  /* case vide supposée à gauche, et passant par dessous */
 { int j;
   for(j=0;j<d;j++)  { descentecv(); droitecv(); droitecv();  monteecv();   gauchecv(); }
 }
void horizontaldroitehaut(int d)  /* case vide supposée à gauche, et passant par dessus */
 { int j;
   for(j=0;j<d;j++) { monteecv(); droitecv(); droitecv();  descentecv(); gauchecv();}
 }
```

Passons maintenant au mouvement global, avec ses composantes verticale et horizontale, grâce à la fonction *mouvement*(*i*, *ifinal*). Comme pour le déplacement de la case vide, on appelle *diffverticale* la distance verticale qui sépare la position actuelle *pos*[*i*] de la case et sa position finale *ifinal*, et de même *diffhorizontale* la distance horizontale. On doit distinguer tous les cas possibles selon que les variables *diffverticale* et *diffverticale* sont positives, nulles ou négatives, et que la variable *flag* vaut 1 ou – 1 (la case vide est à droite ou à gauche de la case).

Nous avons illustré deux cas sur les *figures 9 et 10*. Remarquons que dans le déplacement horizontal, on privilégie de passer par en dessous sauf si l'on se trouve sur la dernière ligne, et que dans le déplacement vertical, on choisit de passer par la droite sauf si l'on est sur la dernière colonne.

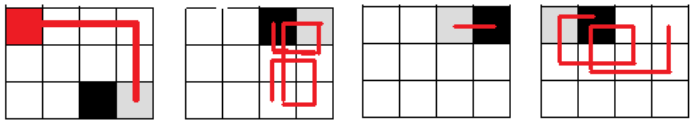
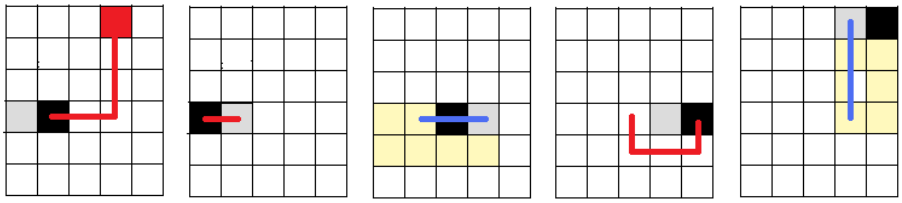


Figure 9 : Cas où *diffverticale* > 0, *diffhorizontale* > 0 et *flag* = – 1 (*cas 3* dans le programme). La case vide est en noir, la case à déplacer doit aller du bloc gris au bloc rouge. On monte, puis on va vers la gauche. Dans le mouvement horizontal, on choisit de passer par en dessous.



Figuree 10 : Cas où *diffverticale* > 0, *diffhorizontale* < 0 et *flag* = 1, la case à déplacer n'étant pas sur la dernière ligne (on passe donc par en dessous), et sa position à atteindre n'est pas sur la dernière colonne (on fera une montée par la droite après avoir déplacé la case vide de gauche à droite), ce qui correspond à un des *cas 5* dans le programme). Ici on doit commencer par le déplacement horizontal.

Comme cela arrive sur la *figure 10*, on est parfois amené à faire subir à la case vide des crochets, comme ici *crochetbddh*() indiquant une descente *b*, puis deux mouvements *d* à droite, et enfin une montée *h*.

```
void crochetbddh(void)
{ descentecv();  droitecv();  droitecv();  monteecv(); }
void crochethddb(void)
{ monteecv(); droitecv(); droitecv(); descentecv(); }
```

On en arrive alors à la fonction *mouvement*(*i*, *ifinal*) :

```
void mouvement(int i,int ifinal)
{ int yfinal,diffverticale,xfinal,diffhorizontale,qq;
  yfinal=ifinal/N;   diffverticale=pos[i]/N-yfinal;
  xfinal=ifinal%N;   diffhorizontale=pos[i]%N-xfinal;

  if (diffverticale>0 && diffhorizontale>0 && flag==1) /********1*******/
       { monteepardroite(diffverticale);   horizontalgauchebas(diffhorizontale);
       }
   else if (diffverticale>0 && diffhorizontale==0 && flag==1) /*****2*******/
        monteepardroite(diffverticale);
  else if (diffverticale>0 && diffhorizontale>0 && flag==-1) /********3*******/
       { monteepargauche(diffverticale);
         droitecv();  horizontalgauchebas(diffhorizontale-1);
       }
    else if (diffverticale>0 && diffhorizontale==0 && flag==-1) /********4*******/
       {  if (pos[i]%N==N-1)
             monteepargauche(diffverticale);
          else if (pos[i]%N<N-1)
            {  if (pos[i]/N==P-1) {crochethddb();  monteepardroite(diffverticale);
                                   }
               else if  (pos[i]/N<P-1)
                  { crochetbddh();  monteepardroite(diffverticale);
                  }
            }
       }
    else if (diffverticale>0 && diffhorizontale<0 && flag==1) /********5*******/
       { if (pos[i]/N==P-1)
           { gauchecv();   horizontaldroitehaut(-diffhorizontale-1);
             if ((pos[i]%N) <N-1)
               {crochethddb(); monteepardroite(diffverticale);
               }
             else if (pos[i]%N==N-1) monteepargauche(diffverticale);
           }
         else if (pos[i]/N<P-1)  /* on n'est pas sur la denière ligne */
           { gauchecv();   horizontaldroitebas(-diffhorizontale-1); /* on passe par en dessous */
             if (pos[i]%N<N-1)
               { crochetbddh(); monteepardroite(diffverticale);} /* on n'est pas sur la dernière colonne */
            else if (pos[i]%N==N-1) monteepargauche(diffverticale); /* on est sur la dernière colonne */
           }
      }
    else if (diffverticale>0 && diffhorizontale<0 && flag==-1) /********6*******/
       { if (pos[i]/N==P-1)
           { horizontaldroitehaut(-diffhorizontale);
             if ((pos[i]%N) <N-1)
                { crochethddb();monteepardroite(diffverticale-1);
                }
             else if (pos[i]%N==N-1) monteepargauche(diffverticale);
           }
         else if (pos[i]/N<P-1)
```

```
        { horizontaldroitebas(-diffhorizontale);
          if (pos[i]%N<N-1)
            { crochetbddh(); monteepardroite(diffverticale);
            }
          else if (pos[i]%N==N-1) monteepargauche(diffverticale);
        }
   }
else if (diffverticale==0 && diffhorizontale<0 && flag==1) /********7*******/
    { if (pos[i]/N==P-1)
         { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(10);;effacer();
           horizontaldroitehaut(-diffhorizontale-1);
         }
       else if (pos[i]/N<P-1)
         { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(10);;effacer();
           horizontaldroitebas(-diffhorizontale-1);
         }
   }
else if (diffverticale==0 && diffhorizontale<0 && flag==-1) /********8*******/
    { if (pos[i]/N==P-1)
         {  horizontaldroitehaut(-diffhorizontale);
         }
       else if (pos[i]/N<P-1)
         { horizontaldroitebas(-diffhorizontale);
         }
   }
else if (diffverticale==0 && diffhorizontale>0 && flag==1)    /**********9******/
    { if (pos[i]/N==P-1)
         horizontalgauchehaut(diffhorizontale);
       else if (pos[i]/N<P-1)
         horizontalgauchebas(diffhorizontale);
     }
  else if (diffverticale==0 && diffhorizontale>0 && flag==-1)    /**********10******/
    { if (pos[i]/N==P-1)
         {droitecv();  horizontalgauchehaut(diffhorizontale-1);
         }
       else if (pos[i]/N<P-1)
         {droitecv();  horizontalgauchebas(diffhorizontale-1);
         }
     }
else if (diffverticale<0 && diffhorizontale>0 && flag==1) /********11*******/
    { descentepardroite(-diffverticale);
       if (pos[i]/N<P-1)
         {  horizontalgauchebas(diffhorizontale);
         }
       else if (pos[i]/N== P-1)  horizontalgauchehaut(diffhorizontale);
     }

else if (diffverticale<0 && diffhorizontale>0 && flag==-1) /********12*******/
    {  descentepargauche(-diffverticale);
       if (pos[i]/N<P-1)
         { droitecv();   horizontalgauchebas(diffhorizontale-1); }
       else  if (pos[i]/N==P-1)
         { droitecv(); horizontalgauchehaut(diffhorizontale-1); }
     }
else if (diffverticale<0 && diffhorizontale<0 && flag==1) /********13*******/
    {  descentepardroite(-diffverticale);
       gauchecv();  horizontaldroitehaut(-diffhorizontale-1);
    }
 else if (diffverticale<0 && diffhorizontale<0 && flag==-1) /********14*******/
    {  descentepargauche(-diffverticale);
```

```
            horizontaldroitehaut(-diffhorizontale);
        }
    else if (diffverticale<0 && diffhorizontale==0 && flag==1) /********15*******/
        descentepardroite(-diffverticale);
    else if (diffverticale<0 && diffhorizontale==0 && flag==-1) /********16*******/
        descentepargauche(-diffverticale);
}
```

Dans tous les cas de figure, on constate qu'un déplacement vertical provoque des perturbations sur deux colonnes, et qu'un déplacement horizontal en provoque sur deux lignes. Cela va induire quelques difficultés supplémentairaires, d'abord pour placer les blocs à la fin de chaque ligne, puis pour les mettre sur les deux dernières lignes.

4.2.3. Remplissage des deux dernières cases de chaque ligne

Il n'est pas possible de remplir la dernière case d'une ligne sans perturber l'avant-dernière case déjà placée. Ainsi, pour les deux dernières cases de chaque ligne, il convient de procéder différemment. Comme on ne peut pas placer les deux blocs carrés l'un après l'autre directement, on va les mettre d'abord tous les deux sur la dernière colonne. Comme par exemple sur la *figure 11*.

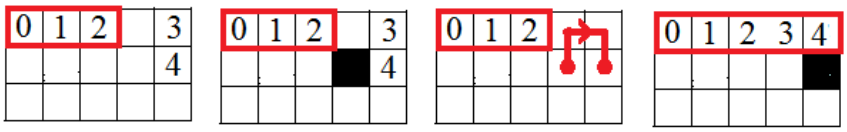


Figure 11 : Dans la première ligne pour $N = 5$, on commence par placer le 3 et le 4 sur la dernière colonne, puis la case vide fait une boucle pour mettre le 3 et le 4 dans leur position définitive.

Plaçons-nous sur une ligne numérotée par la variable *ligne* (de 0 à $P - 1$). La position de la dernière case de la ligne est notée *a1* et celle de l'avant-dernière case est notée *a2*. Par exemple $a1 = 4$ et $a2 = 3$ sur la *figure 11*. L'objectif intermédiaire[3] est de mettre le bloc carré *a2* en position *a1* et le bloc *a1* en position $a1 + N$, comme le sont le 3 et le 4 sur la *figure 11 à gauche*.

Au tout début des opérations, la case vide se trouve toujours en position $N - 2$ (juste à droite du 2 qui vient d'être placé, dans le contexte de la *figure 11*). Si les deux blocs carrés sont déjà dans leur position intermédiaire, comme sur la *figure 12*, il ne reste qu'à déplacer d'un cran à droite la case vide, puis à la descendre d'un cran pour placer les deux blocs carrés dans leur position définitive.

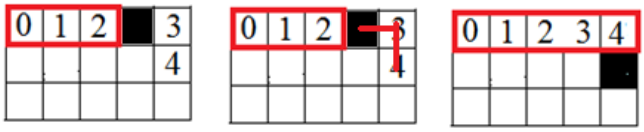


Figure 12 : *A gauche*, cas particulier où les cases 3 et 4 sont déjà dans leur position intermédiaire. *A droite* la situation finale

Mais sinon ? Il va d'abord falloir placer le bloc *a2* en *a1* (dans notre exemple, le 3 dans sa position intermédiaire sur la dernière colonne), et ensuite le bloc *a1* en $a1 + N$ (le 4 dans sa position intermédiaire sur la dernière colonne). Mais il y a un problème. Une fois le 3 bien placé, avec la case vide juste à sa gauche, il peut arriver que le 4 soit juste au-dessous, et l'on aura soit l'impossibilité de pouvoir le placer dans sa position intermédiaire, soit son propre déplacement va faire rebouger le 3. Ainsi lorsqu'au début des opérations le bloc *a1* (4 dans notre exemple) se trouve dans une des trois positions *a1*, $a1 + N$ ou $a1 + N - 1$ (*figure 13*), on décide de le déplacer en premier lieu hors de cette zone, en le plaçant en position $a1 + N - 2$.

[3] Cela explique a posteriori pourquoi la fonction *mouvement* possède deux variables, la deuxième étant *ifinal*. On ne fait plus directement *mouvement*(*i*, *i*) lorsqu'il convient de passer d'abord par une situation intermédiaire comme dans le cas présent.

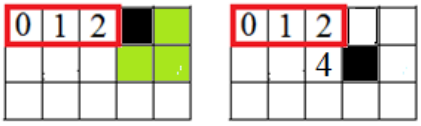


Figure 13 : Si le 4 se trouve dans l'une des trois cases vertes au début des opérations (*à gauche*) on commence par le sortir de cette zone en le mettant juste au-dessous du 2 (*à droite*)

Cela étant fait, du moins si le cas se présente, on peut alors placer le 3 dans sa position intermédiaire puis le 4 aussi. La case vide se trouve alors juste à gauche du 4. Il ne reste plus qu'à faire la boucle qui met le 3 et le 4 dans leur position définitive (*cf. figure 11*), grâce à la fonction *crochetfinal*() :

```
void crochetfinal(void)
{ monteecv();dessin();SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
  droitecv(); dessin();SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
  descentecv();dessin();SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
}
```

Au stade où nous en sommes, nous pouvons commencer à faire le programme principal, qui va permettre de placer tous les blocs dans leur bonne position, sauf pour les deux dernières lignes, car le même problème va se poser pour celles-ci que le problème que l'on a eu pour remplir les deux dernières cases de chaque ligne.

```
/* conditions initiales */
permutation();  /* désordre initial */
dessin();  SDL_Flip(screen); pause();SDL_FillRect(screen,0,blanc);
/* remplissage de chaque ligne, sauf ses deux dernières cases */
for(ligne=0;ligne<P-2;ligne++)
  { for(i=ligne*N+0;i<ligne*N+N-2;i++)
     if (a[i]!=i) {casevidecolle(i);     mouvement(i,i);   }
    a2=ligne*N+N-2;   a1=ligne*N+N-1; /*  les deux dernières cases d'une ligne */
    if (a[a1]==a2 && a[a1+N]==a1)  /* blocs a1 et a2 déjà en position sur la dernière colonne */
        { droitecv(); descentecv();}
    else
      { if ( pos[a1]==a1 || pos[a1]==a1+N || pos[a1]==a1+N-1) /* dans ces cas, on sort a1 de la zone */
          { casevidecolle(a1);   mouvement(a1,a1+N-2); }
        if (pos[a2]!=a1)  /* placement du bloc a2 en a1 */
          { casevidecolle(a2);   mouvement(a2,a1); }
        if (pos[a1]!=a1 ) /* placement du bloc a1 en a1 + N */
          { casevidecolle(a1);    mouvement(a1,a1+N);  }
        crochetfinal();
      }
  }
```

On arrive ainsi à un résultat comme celui de la *figure 14*.

14	2	12	0	18
3	22	7	10	20
13	5	1	4	23
19	6	21	8	11
17	15	9	16	

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
17	20	23	22	
21	16	15	18	19

Figure 14 : *A gauche*, la permutation initiale, ici pour $N = P = 5$, *à droite* la remise en ordre partielle, les deux dernières lignes n'étant pas encore traitées

4.2.4. Remplissage des deux dernières lignes

Il reste à traiter les deux dernières lignes, numérotées $P - 2$ et $P - 1$, avec au départ la case vide tout à droite sur la colonne $P - 2$ (*cf. figure 14*). Nous allons commencer par placer en position définitive les deux éléments de la première colonne (colonne numéro $i = 0$), puis ceux de la deuxième colonne, et ainsi de suite jusqu'aux deux dernières colonnes, où il suffira de faire une boucle pour arriver à l'ordre final avec la case vide en bas à droite. A chaque étape la case vide doit se trouver sur la case haute de la colonne à traiter, ce qui nécessite au départ de la déplacer de droite à gauche. On va obtenir la succession d'étapes indiquée sur la *figure 15*.

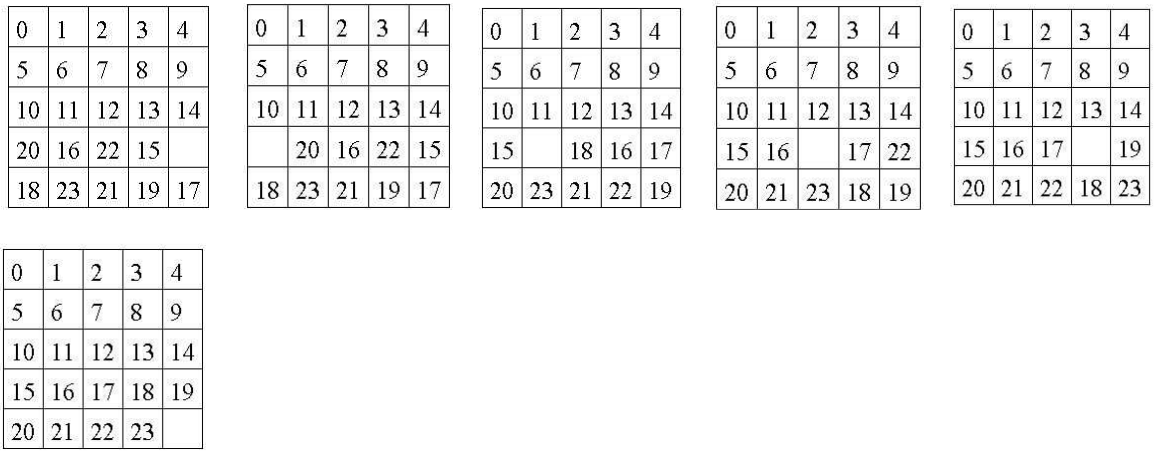


Figure 15 : *En haut à gauche*, la situation initiale avec seulement les deux dernières lignes dans le désordre, puis sur le deuxième dessin on commence par placer la case vide à gauche. Sur les trois figures suivantes *en haut*, on met en bonne position les deux éléments de la première colonne, puis ceux de la deuxième, et enfin ceux de la troisième. *En bas*, on fait un cycle final pour mettre les deux dernières colonnes dans l'ordre.

Plaçons-nous sur la colonne i (avec i de 0 à $N - 3$). Il s'agit de remplir les cases *a1* et *a2* de cette colonne (*figure 16*).

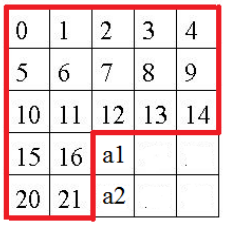


Figure 16 : Position des cases *a1* et *a2* à remplir, ici sur la colonne $i = 2$ pour $N = P = 5$. Il s'agit de placer le bloc *a1* = 17 en position *a1*, et le bloc *a2* = 22 en *a2*

Prenons l'exemple le plus simple avec $N = 4$, $P = 2$ et $i = 0$. Il s'agit de mettre le 0 et le 4 sur la première colonne. Mais cela ne peut pas être fait directement. Il convient de passer par une étape intermédiaire où le 0 et le 4 sont mis l'un après l'autre sur la dernière ligne, c'est-à-dire le bloc *a1* en position *a2* et le bloc *a2* en position *a2* + 1, suivant la configuration de la *figure 17 à gauche*. Il ne restera plus qu'à faire un mouvement tournant de la case vide pour arriver à la configuration définitive (*figure 17 à droite*), cela grâce à la fonction *crochet*() :

```
void crochet(void)
{ monteecv();gauchecv();gauchecv();descentecv();droitecv();monteecv();
}
```

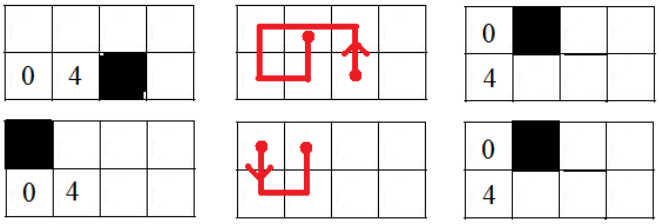


Figure 17 : *En haut à gauche*, la configuration intermédiaire à obtenir dans le cas général où le 4 a été déplacé, d'où la case vide à sa droite, *au centre* le crochet effectué par la case vide, *à droite* la configuration définitive. En bas, le cas particulier où le 0 et le 4 sont bien placés d'entrée de jeu, avec la case vide en haut à gauche, ce qui nécessite un petit mouvement de la case vide pour arriver à la position finale.

On constate qu'à l'issue de ce placement, la case vide se trouve en haut à gauche de la colonne suivante à traiter. Cela explique a posteriori pourquoi on a placé la case vide en haut à gauche avant le début des opérations sur la première colonne.

Mais pour arriver à cette configuration intermédiaire, il convient de faire attention à la position initiale du bloc *a2*. Si celui-ci se trouve dans l'une des trois cases de gauche qui entourent la case vide (*figure 18*), il faudra d'abord l'éloigner en le plaçant en position *a1* + 2, afin de pouvoir placer le bloc *a1* dans sa bonne position, sans crainte de perturbations lorsque le bloc *a2* devra être placé à son tour.

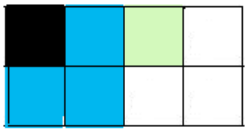


Figure 18 : En noir la case vide au départ. Si le bloc 4 se trouve dans l'une des trois cases en bleu, on l'envoie dans la position mise en vert

Enfin, si le 4 se trouve au départ dans la ligne du bas, avec le 0 déjà bien placé, le mouvement de la case vide doit d'abord se faire horizontalement puis verticalement (*figure 19*), car si l'on commençait par un mouvement vertical, le 0 serait déplacé.

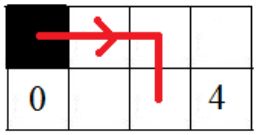


Figure 19 : Déplacement de la case vide dans ce cas particulier, avant le placement du 4

On agit ainsi pour chacune des colonnes numérotées *i*, avec *i* allant de 0 à *N* – 3. Il reste alors les deux dernières colonnes, où se trouvent les trois derniers blocs à placer ainsi que la case vide. La fonction *cyclefinal*() se charge du placement final, en distinguant les trois situations possibles (*figure 20*).

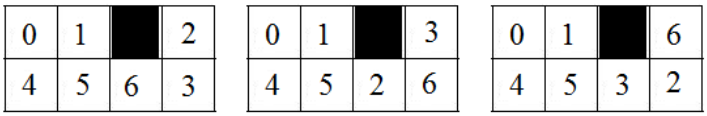


Figure 20 : Dans l'exemple où *N* = 4 et *P* = 2, les trois situations possibles pour les deux dernières colonnes, avant de faire agir la fonction *cyclefinal*()

```
void cyclefinal(void)
{ droitecv(); /* on envoie la case vide à droite */
  if (pos[N*(P-2)+N-2]==N*(P-2)+N-2) descentecv(); /* premier cas */
  else  if (pos[N*(P-2)+N-1]==N*(P-2)+N-2) { gauchecv(); descentecv();droitecv();} /* deuxième cas */
   else if (pos[N*(P-2)+2*N-2]==N*(P-2)+N-2)    /* troisième cas */
```

```
      {descentecv();gauchecv();monteecv();droitecv(); descentecv();}
  dessin();SDL_Flip(screen); SDL_Delay(10);
 }
```

On en arrive ainsi à la suite et fin du programme principal :

```
for(i=0;i<N-1;i++) gauchecv();  /* case vide mise à gauche */
for(i=0;i<N-2;i++)  /* toutes les colonnes i sauf les deux dernières */
  { a1=N*(P-2)+i;
    a2=N*(P-2)+N+i;
    if (pos[a1]==a2 && pos[a2]==a2+1) {descentecv();droitecv();monteecv();} /* cas particulier */
    else
      { if (pos[a2]==a2 || pos[a2]==a2+1 || pos[a2]==a1+1)  /* cas où le bloc a2 doit être éloigné */
          { casevidecolle(a2);mouvement(a2,a1+2);
            dessin(); SDL_Flip(screen);SDL_FillRect(screen,0,blanc);
          }
        if (pos[a1]!=a2)  /* si ce n'est pas déjà fait on place a1 en position a2 */
          {casevidecolle(a1);   mouvement(a1,a2);
           dessin(); SDL_Flip(screen);SDL_FillRect(screen,0,blanc);
          }
        if(pos[a2]!=a2+1) /* si ce n'est pas déjà le cas, on place a2 en position a2 + 1 */
          {casevidecolle(a2);mouvement(a2,a2+1);
           dessin(); SDL_Flip(screen);SDL_FillRect(screen,0,blanc);
          }
        crochet();dessin(); SDL_Flip(screen);SDL_FillRect(screen,0,blanc);
      }
  }
cyclefinal();
```

5. Utilisation de cycles d'ordre 3

On sait qu'une permutation paire, condition nécessaire du jeu de taquin, peut être engendrée par les décalages cycliques d'ordre 3, de la forme (i – 2, i – 1, i). Cela signifie que l'élément en position i – 2 va en position i – 1, celui en position i – 1 va en position i, et celui en position i va en position i – 2, sans qu'aucun des autres éléments ne bouge.

Mieux encore, il suffit de prendre les décalages cycliques d'ordre 3 en chaîne pour engendrer les permutations paires. Par chaîne, nous voulons dire que le troisième élément d"un cycle est le premier élément du suivant. Autrement dit, toute permutation paire est engendrée par les cycles (0 1 2), (2 3 4), (4 5 6), (6 7 8), etc. Numérotons chaque cycle par son dernier élément. Les décalages cycliques chaînés sont ainsi numérotés 2, 4, 6, 8, etc., avec un éventuel petit problème à la fin. Si *NP* est pair, avec la permutation portant sur les éléments allant de 0 à *NP* – 2 (la case vide étant *NP* – 1), le chaînage des cycles est parfait, puisque le dernier décalage porte un numéro pair *NP* – 2. Mais si *NP* est impair, l'élément *NP* – 2 est impair, et il convient d'ajouter aux décalages à numéros pairs le décalage cyclique numéroté *NP* – 2, qui empiète sur le cycle précédent avec deux éléments au lieu d'un (*figure 21*).

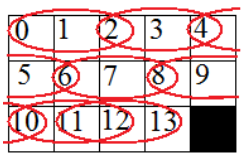


Figure 21 : Tableau des positions lorsque *N* = 5 et *P* = 3. Les décalages cycliques d'ordre 3 (encadrés en rouge) sont numérotés 2, 4, 6, 8, 10, 12, et il convient d'ajouter le décalage final 13 qui empiète sur deux éléments de son prédécesseur, puisque *NP* est impair

Mais comment effectuer de tels décalages cycliques dans le cadre du jeu du taquin, sans qu'aucun des autres éléments ne bouge ? On doit distinguer trois cas selon que les trois éléments du cycle sont sur la même ligne, ou que le cycle empiète sur deux lignes.

Premier cas : Le cycle a ses trois éléments sur la même ligne. Il s'agit de réaliser le cyle (*i* – 2, *i* – 1, *i*) qui provoque un décalage cyclique vers la droite. Mais on peut aussi pratiquer le cycle inverse (*i*, *i* – 1, *i* – 2,) qui provoque un décalage cyclique vers la gauche. Un exemple est donné sur la *figure 22*.

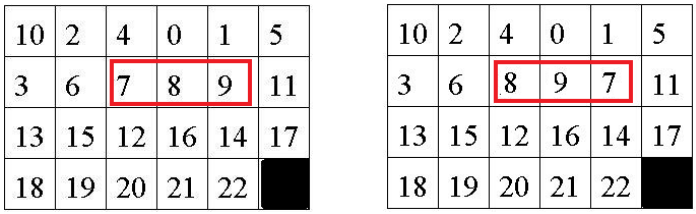


Figure 22 : Réalisation du cycle (10 9 8) pour *N* = 6, ce qui provoque un décalage cyclique vers la gauche des éléments 7, 8, 9 (encadrés en rouge)

Cela se fait en trois temps :

1) La case vide, située au départ en bas à droite (position *NP* – 1) est envoyée sous la case en position *i*; à condition que la case ne soit pas sur la dernière ligne. Pour cela on fait d'abord un déplacement vertical, puis un déplacement horizontal (*figure 23 en haut*).

2) On effectue trois cycles successifs, d'abord *hggbdd* (*h* pour haut, *g* pour gauche, *b* pour bas, *d* pour droite), puis *ghdb*, et enfin *gghddb* (cycle à l'envers du premier), comme sur la *figure 23 au centre*. La succession de ces trois cycles revient à faire *hggbdhdbgghddb*. Cela provoque le décalage cyclique à gauche, et cela remet en place les autres cases qu'il a fallu bouger.

3) La case vide est remise dans sa position en bas à droite, en pratiquant un déplacement horizontal puis un déplacement vertical, ce qui remet tout à sa place initiale, sauf les trois éléments du cycle.

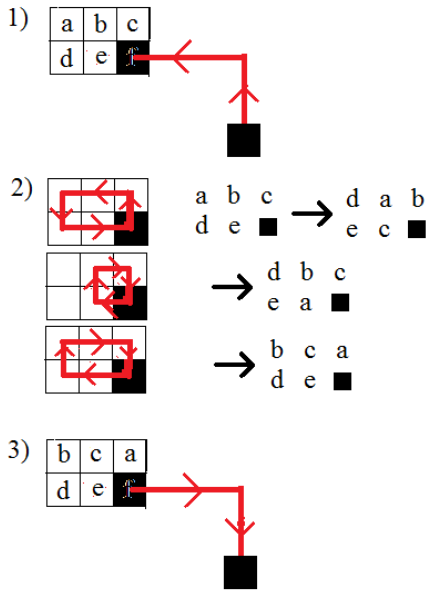


Figure 23 : Réalisation d'un décalage à gauche faisant passer de *a b c* à *b c a*, sans qu'aucune autre case n'ait bougé

Passons à la programmation. Pour le déplacement de la case vide (*cas 1)* et *3)*), on utilise la fonction *casevideversposition*(int *i*) qui envoie la case vide vers la position *i*, soit avec un déplacement

vertical vers le haut suivi d'un déplacement horizontal vers la gauche, soit avec un déplacement horizontal vers la droite suivi d'un déplacement vertical vers le bas.

```
void casevideversposition(int i)
 { int diffverticale,diffhorizontale,j,xfinale,yfinale,q;
    xcv=poscv%N; ycv=poscv/N;
    xfinale=i%N; yfinale=i/N;
   diffverticale=ycv-yfinale;
   diffhorizontale=xcv-xfinale;
   if (diffverticale>=0 && diffhorizontale>=0)
     { for(j = 0;j<diffverticale;j++)
         { monteecv(); dessin();  SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc); }
       for(j = 0;j<diffhorizontale;j++)
         { gauchecv();dessin();SDL_Flip(screen); SDL_Delay(10);;SDL_FillRect(screen,0,blanc); }
      }
    else if (diffverticale<=0 && diffhorizontale<=0)
     { for(j = 0;j<-diffhorizontale;j++)
         { droitecv();dessin();SDL_Flip(screen); SDL_Delay(10);;SDL_FillRect(screen,0,blanc); }

       for(j = 0;j<-diffverticale;j++)
         { descentecv(); dessin();  SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc); }
      }
}
```

Pour les cycles effectués par la case vide à l'étape 2, provoquant le cycle vers la gauche, on construit la fonction *cycle3*(), qui utilise les fonctions de montée, descente, de mouvement à droite et à gauche, précédemment définies.

```
void cycle3(void)
{ monteecv();gauchecv();gauchecv();descentecv();droitecv();
  monteecv();droitecv();descentecv();
  gauchecv();gauchecv();monteecv();droitecv();droitecv();descentecv();
}
```

Dans le cas où le cycle à réaliser se trouve sur la dernière ligne, on modifie légèrement la fonction, notamment en envoyant la case vide au-dessus et non plus en dessous :

```
void cycle3dernierligne(void)
{ descentecv();gauchecv();gauchecv();monteecv();droitecv();
  descentecv();droitecv();monteecv();
  gauchecv();gauchecv();descentecv();droitecv();droitecv();monteecv();
}
```

Puis on regroupe ces fonctions pour la fonction *cyclegpositionfinale*(int *i*) qui traite le décalage cyclique vers la gauche lorsque le dernier élément du cycle est en position *i*, soit :

```
void cyclegpositionfinale(int i)
{ if (i<(P-1)*N)
      {casevideversposition(i+N);cycle3();casevideversposition(NP-1);}
   else
      {casevideversposition(i-N);cycle3dernierligne();casevideversposition(NP-1);}
  dessin();  SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
}
```

Pour provoquer un décalage cyclique vers la droite, il suffit de faire deux fois un décalage vers la gauche :

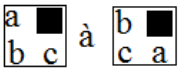
```
void cycledpositionfinale(int i)
{casevideversposition(i+N);cycle3();casevideversposition(NP-1);
```

```
 casevideversposition(i+N);cycle3();casevideversposition(NP-1);
 dessin();  SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
}
```

Remarquons que cela vaut aussi bien pour les cycles chaînés générateurs (0 1 2) (2 3 4), etc. que pour tout cycle ($i-2$, $i-1$, i) ayant ses éléments sur la même ligne.

Deuxième cas : Le cycle empiète sur deux lignes, avec un élément à droite sur la ligne du dessus et les deux autres à gauche sur la ligne du dessous. Remarquons que ce cas ne se présente que si N est impair.

Quand une telle situation se ptoduit, cela signifie que la ligne du haut est formée d'une chaîne de décalages cycliques. Comme par exemple sur la *figure 24-a)*, avec $N = 7$, où l'on a la chaîne 0 1 2 3 4 5 6. En faisant une succession, de la droite vers la gauche, de ces cycles à droite, il se produit un cycle à droite de l'ensemble de la ligne, et le dernier élément est envoyé en premier (*figure 24-b*). Rappelons que l'on sait faire ce genre de cycles, grâce au premier cas que nous avons traité, avec à chaque fois la case vide renvoyée dans sa case en bas à droite. Puis on envoie la case vide juste à droite du premier élément de la ligne, comme on sait le faire (*figure 24-c*), et en faisant le cycle *gbdh*, les trois éléments de notre cycle font un mouvement tournant (*figure 24-d*) :

passage de  à

Ensuite la case vide fait le trajet inverse pour retourner dans sa case en bas à droite, ce qui remet en place les éléments qui avaient bougé (figure 24-e)? Enfin, on refait la succession des cycles de la ligne, mais cette fois les cycles à gauche, et de gauche à droite (*figure 24-f*), pour remettre les éléments à leur place, et l'on a bien le cycle à gauche que l'on voulait.

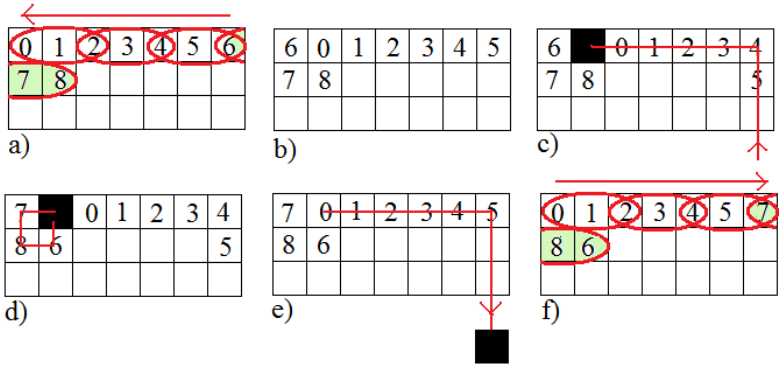


Figure 24 : Réalisation d'un cycle à gauche pour $N = 7$, où le bloc 678 est transformé en 786, sans que rien d'autre ne bouge

On en déduit la fonction associée, notée *cycle12g*(i) où i est la position du dernier élément du cycle à faire :

```
void cycle12g(int i)
{  int j;
   for(j=0;j<(N-1)/2;j++)  cycledpositionfinale(i-2-2*j);  /* cycles à droite sur toute la ligne */
   casevideversposition(i-N);  /* déplacement de la case vide */
   gauchecv();descentecv();droitecv();monteecv(); /* cycle de la case vide */
   casevideversposition(NP-1);  /* renvoi de ma case vide en bas à droite */
   dessin();  SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
   for(j=(N-1)/2-1;j>=0;j--)  cyclegpositionfinale(i-2-2*j);  /* cycles à gauche sur toute la ligne */
}
```

Troisième cas : Le cycle empiète sur deux lignes, avec deux éléments à droite sur la ligne du dessus et le dernier à gauche sur la ligne du dessous. Ce cas se présente pour *N* impair (sur une ligne sur deux) ou pair (sur chaque ligne). On doit distinguer ces deux éventualités.

a) *N* est impair : la ligne du dessus peut être entièrement chaînée, à partir de la position *i* – 1 à droite. Remarquons que les cycles ne sont pas ceux de la chaîne génératrice (0 1 2) (2 3 4), etc., mais nous les traitons par la méthode du premier cas.[4] On commence par faire la succession, de droite à gauche, de ces cycles à droite, et cela deux fois de suite, ce qui amène les deux derniers éléments de la ligne au début de la ligne (*figure 25*). Puis on amène la case vide en position *i* + 1, de façon à pratiquer le mouvement tournant sur les trois éléments concernés du cycle. Enfin, on refait deux fois la succession, de gauche à droite, des cycles de la ligne supérieure, pour arriver au résultat recherché.

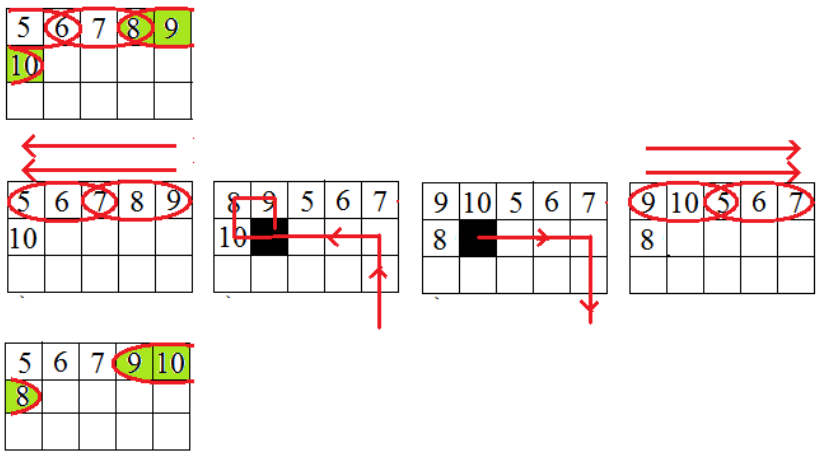


Figure 25 : *En haut*, la situation initiale, dans le cas où *N* = 5, la ligne concernée étant la deuxième dans le rectangle du taquin, avec le bloc 8 9 10 que nous voulons transformer. *Au centre* sont indiquées les opérations successives à effectuer. *En bas* on aboutit au bloc 9 10 8, correspondant à un cycle à gauche

b) *N* est pair : Une légère modification doit être opérée. Prenons par exemple *N* = 6, avec la ligne supérieure 0 1 2 3 4 5, le cycle à effectuer concernant les trois éléments 4 5 6. On commence par envoyer le 4 (élément *i* – 2) en premier, en faisant la succession de cycles à droite, à partir de *i* – 2, de droite à gauche : 0 1 2 3 4 5 ce qui donne 4 0 1 2 3.

Puis on envoie l'élément 5 en second en faisant la succession de cycles, cette fois-ci, à partir de la position *i* – 1 : 4 0 1 2 3 5 . On aboutit à la configuration voulue sur la ligne supérieure :
4 5 0 1 2 3.

Pour le reste on fait comme dans le cas où *N* est impair.

On aboutit à la fonction *cycleg21*(*i*), et si l'on veut visualiser les déplacements successifs, il conviendra d'ajouter à chaque étape une ligne du style :

```
dessin();  SDL_Flip(screen); SDL_Delay(10);SDL_FillRect(screen,0,blanc);
```

où l'on reprendra la fonction *dessin*() programmée précédemment.

```
void cycle21g(int i)
{  int j;
```

[4] Faisons une digression théorique : tout cycle (*i* – 2, *i* – 1, *i*) se déduit de la chaîne génératrice (0 1 2) (2 3 4), etc. Comment obtenir (1 2 3) par exemple ? On remarque que (2 3 4) (0 1 2) = (0 1 2 3), et en inversant on a (3 2 1 0). Il suffit alors de faire (3 2 1 0) (0 1 2) (0 1 2 3) pour obtenir (1 2 3). On dit que (1 2 3) est le conjugué de (0 1 2) par (0 1 2 3).

```
   if (N%2==1)  /* cas où N est impair */
     { for(j=0;j<(N-1)/2;j++)  cycledpositionfinale(i-1-2*j);
       for(j=0;j<(N-1)/2;j++) cycledpositionfinale(i-1-2*j);
       casevideversposition(i+1);
       gauchecv();monteecv();droitecv();descentecv();
       casevideversposition(NP-1);
       for(j=(N-1)/2-1;j>=0;j--) cyclegpositionfinale(i-1-2*j);
       for(j=(N-1)/2-1;j>=0;j--) cyclegpositionfinale(i-1-2*j);
     }
   else   /* N pair */
    { for(j=0;j<(N-2)/2;j++)  cycledpositionfinale(i-2-2*j);
      for(j=0;j<(N-2)/2;j++) cycledpositionfinale(i-1-2*j);
       casevideversposition(i+1);
       gauchecv();monteecv();droitecv();descentecv();
       casevideversposition(NP-1);
       for(j=(N-2)/2-1;j>=0;j--) cyclegpositionfinale(i-1-2*j);
       for(j=(N-2)/2-1;j>=0;j--) cyclegpositionfinale(i-2-2*j);
     }
}
```

Au stade où nous en sommes, nous sommes en mesure de faire n'importe quel cycle à gauche parmi ceux de la chaîne (0 1 2) (2 3 4), etc., dans tous les cas de figure. Regroupons les trois cas dans une seule fonction *cycle*(*i*) :

```
void cycle(int i)
{
   if ((i-1)%N==0) cycle12g(i);
   else if ( i%N==0) cycle21g(i);
   else cyclegpositionfinale(i);
}
```

Il reste à mettre en place le scénario glonal. La méthode la plus simple consiste à faire d'abord une permutation paire dans le tableau du taquin. Pour cela, on part des blcos carrés disposés dans l'ordre naturel. Puis on pratique un certain nombre de cycles à droite choisis au hasard parmi les cycles chaînés (0 1 2), (2 3 4), (4 5 6), etc. dont on sait qu'ils génèrent une permutation paire quelconque. C'est ce que fait la fonction *permutation*(), avec l'appoint de la fonction *creercycled*(*i*) qui crée un cycle à droite :

```
void permutation(void)
{ int i,j,k,hasard;
  for(i=0;i<NP-1;i++) a[i]=i;   /* les blocs carrés dans l'ordre naturel */
   nbcycles=20;  /* on se donne un certain nombre de cyles à faire */
   k=0;
   for(i=2;i<NP-1;i+=2) h[k++]=i; /* dans le tableau h[], les positions i paires de tous les cycles chaînés */
   if (NP%2==1) h[k++]=NP-2;  /* le cycle supplémentaire lorsque NP est impair */
   for(j=0;j<nbcycles;j++)
     { hasard=rand()%k; /* tirage au hasard d'un cycle numéro i */
       c[j]=h[hasard];  /* enregistrement du cycle dans le tableau c[] */
     }
   for(j=0;j<nbcycles;j++)  creercycled(c[j]);   /* création du décalage cyclique à droite */
   a[NP-1]=NP-1;
   for(i=0;i<NP-1;i++) pos[a[i]]=i;  /* tableau des positions */
   poscv=NP-1; /* posiiton de la case vide */
}

void creercycled(int i)
{  int c1,c2,c3;
   c1=a[i-2];c2=a[i-1];c3=a[i];
   a[i-1]=c1; a[i]=c2;a[i-2]=c3;
```

```
}
```

Le programme principal se réduit à faire la permutation, puis à réaliser les cycles à gauche en parcourant à l'envers le tableau *c*[] des cycles à droite. On revient ainsi à l'ordre naturel.

```
permutation();dessin();  SDL_Flip(screen); pause();SDL_FillRect(screen,0,blanc);
for(i=0;i<nbcycles;i++) cycle(c[nbcycles-1-i]);
dessin();  SDL_Flip(screen); pause();
```

Un résultat du programme est donné sur la *figure 26*.

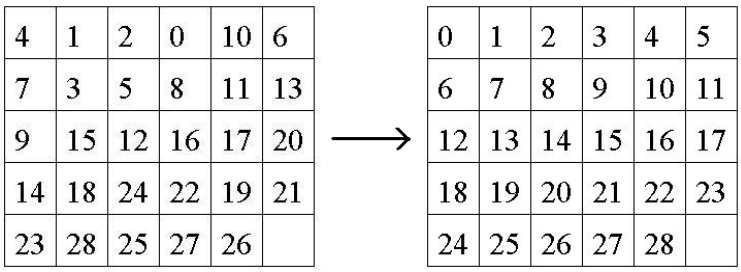


Figure 26 : *A gauche* la permutation initiale pour $N = 6$ et $P = 5$, *à droite* le résultat final après de multiples déplacements de la case vide du taquin

L'intérêt de cette méthode est essentiellement théorique. En effet, une fois démontré que la permutation initiale doit être paire, sans quoi la remise en ordre est impossible,[5] il reste à prouver l'essentiel, à savoir que par le seul jeu des déplacements de la case vide, on peut arriver à cette remise en ordre. C'est là que l'on a utilisé le fait qu'une permutation paire est engendrée par le chaînage de décalages d'ordre 3, (0 1 2), (2 3 4), (4 5 6), etc., et nous avons montré ici que n'importe lequel de ces décalages pouvait être effectué par les mouvements de la case vide.

Par contre, l'aspect pratique est moins intéressant. En effet après chaque opération de décalage nous forçons la case vide à revenir à sa place privilégiée en bas à droite, ce qui provoque de multiples déplacements. La remise en ordre est donc longue, même en prenant seulement quelques dizaines de cycles pour fabriquer la permutation initiale, ce qui limite le désordre initial, comme on peut le vérifier sur la *figure 26 à gauche*.

[5] cf. Pierre Audibert, *Combien ? Mathématiques appliquées à l'informatique*, volume 1, éditions Hermès 2008. Et aussi N. Rifaai : *Jeux de permutations, jeu de taquin*, mémoire de maîtrise informatique, Université Paris 8, 2006.