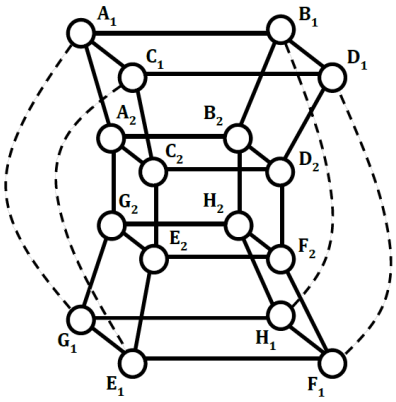
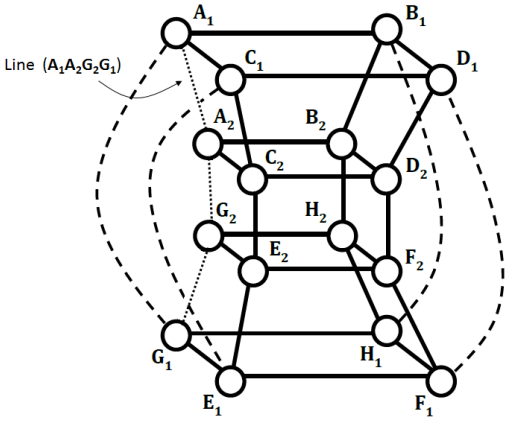
The point-line structure of Tesseract can be thought of as a data model where 16 vertices and 32 edges, organized in a specific way, form a 3-dimensional closed multigraph. The data structure is utilized to generate symmetric key transposition ciphers by assigning each vertex a data value and then performing different operations (e.g. plane rotations) on the structure.

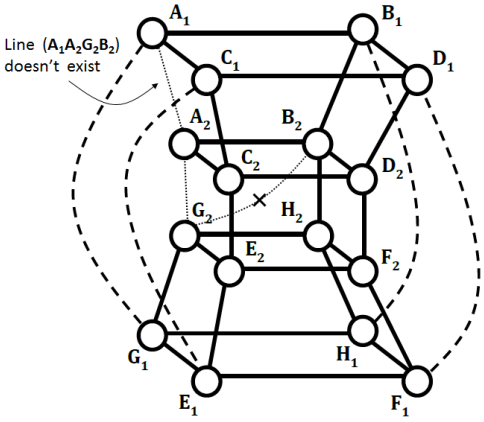
The following vertex nomenclature is used for the rest of this document.



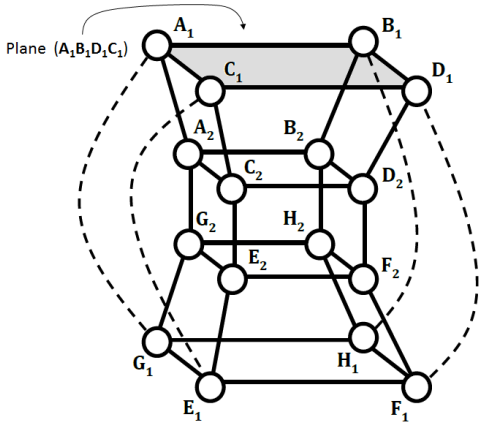
A Tesseract consists of 16 vertices, 32 edges, 24 planes and 8 cubes. In this document, we use only 3 basic structural components of a Tesseract for the first two (out of total three) phases of encryption:

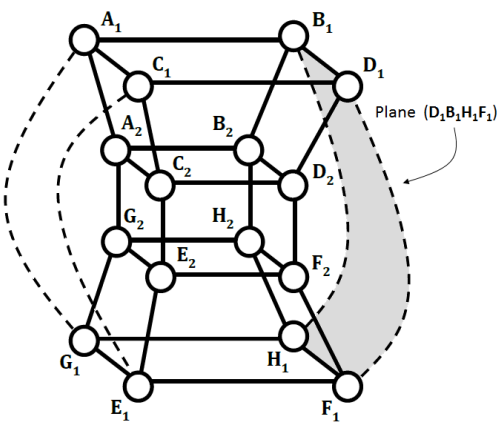
- Vertex – each of the 16 vertices is identified with a label (like **A_1**, **E_2**, **H_2** etc.). They are used for storing data elements.
- Line – any path within a Tesseract is termed as a line. E.g., suppose there is an edge between a pair of vertices v_1 and v_2, and another between v_2 and v_3. Hence, v_1v_2, v_2v_3, $v_1v_2v_3$ are 3 lines in the Tesseract (all 3 are coincident in this case).
  In the above figure, **$A_1A_2G_2G_1$** represents a line, but **$A_1A_2G_2B_2$** does not, as there is no edge between **G_2** and **B_2**.





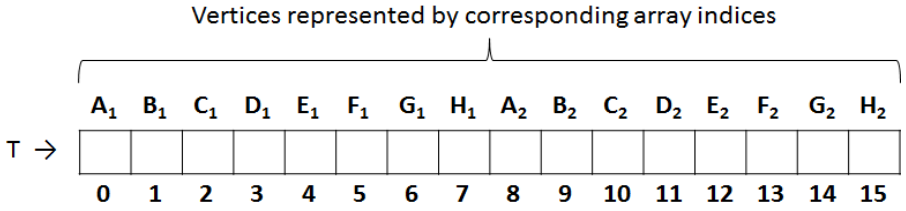
- Plane – a cycle consisting of 4 vertices is called a plane. A Tesseract has 24 planes all of which are squares in shape. A plane can be identified by taking all 4 of its enclosing vertices in order. E.g., **$A_1B_1D_1C_1$** represents a plane, **$D_1B_1H_1F_1$** another.





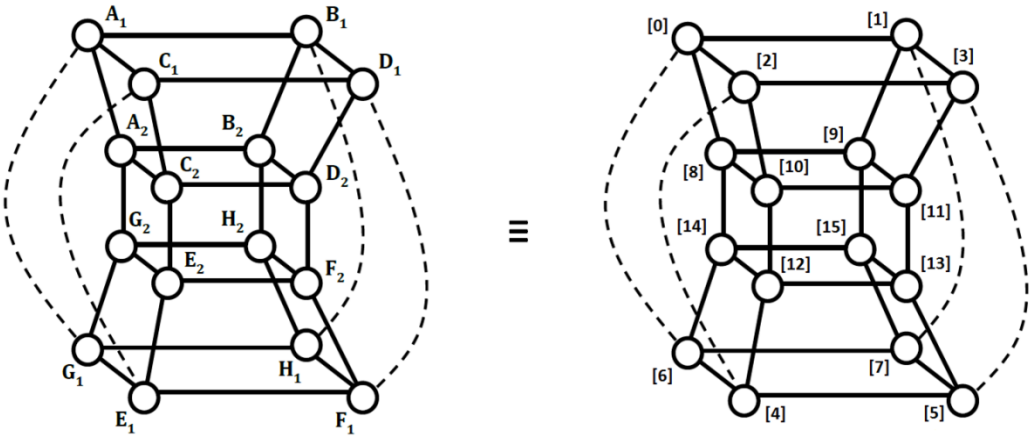
This logical data model is implemented using an array **T**, which represents the set of vertices, and a matrix **M**, representing the set of planes.

The array **T** is a 16 element array. Each index of this array represents a distinct vertex of the Tesseract, and the data element at that index specifies the value therein. We use the following mapping convention for the rest of this document to represent the set of vertices of a Tesseract using array **T**.

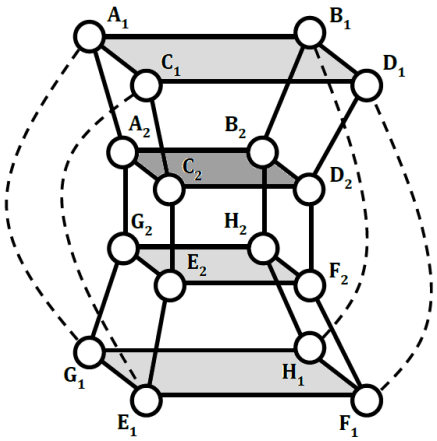


To assign or retrieve some data value to/from some vertex, say $\mathbf{A_1}$, we will refer to its corresponding index number, which is **0** in this case (from the above figure), and perform the specified operation on **T [0]**. Similarly, to access the value at vertex $\mathbf{D_1}$, we refer to its index number **3**, and operate on the element stored at **T [3]**. Hence each vertex of a Tesseract can be referred to using an integer (index no.) between 0 and 15.

Here in this document, the name labels and index numbers of the vertices are used interchangeably. E.g., *'vertex* $\boldsymbol{B_1}$*'* and *'vertex **1**'* refer to the same point. The planes can also be represented the same way using index numbers of their 4 vertices instead of name labels. E.g., 'plane $\mathbf{A_1B_1D_1C_1}$' and 'plane **(0, 1, 3, 2)**' are synonymous.

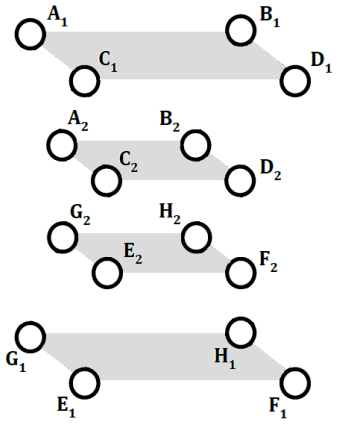


Every single plane in a Tesseract has 3 other planes parallel to it. E.g., for the plane $\mathbf{A_2B_2D_2C_2}$, there are the three other planes $\mathbf{A_1B_1D_1C_1}$, $\mathbf{G_2H_2F_2E_2}$, $\mathbf{G_1H_1F_1E_1}$ which are parallel to it. These 4 planes altogether form a set. A Tesseract contains a total of 24 planes, which can be thought of as a collection of 6 such sets, each containing 4 parallel planes.

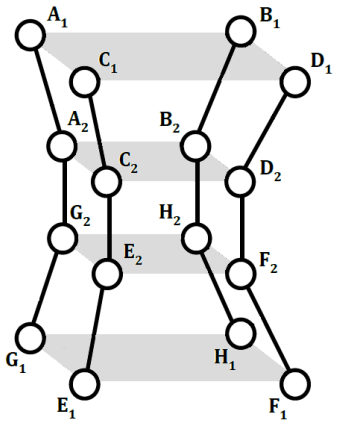


Two important characteristics of parallel planes are –

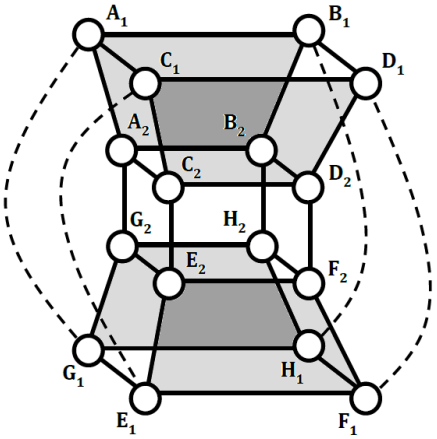
i. They do not share common vertices. The previous example shows a certain set of 4 parallel planes, each containing 4 vertices none of which belongs to any other plane in the same set.



ii. Every vertex in a plane is connected to 3 other vertices, each belonging to a different plane of the other 3 parallel planes, with a 4-point line.

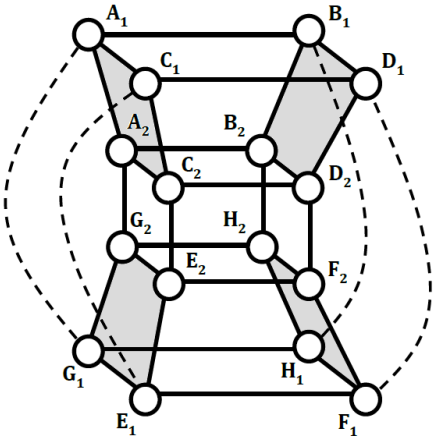


The 6 sets each composed of 4 parallel planes are given below. They collectively construct the 24-plane model.



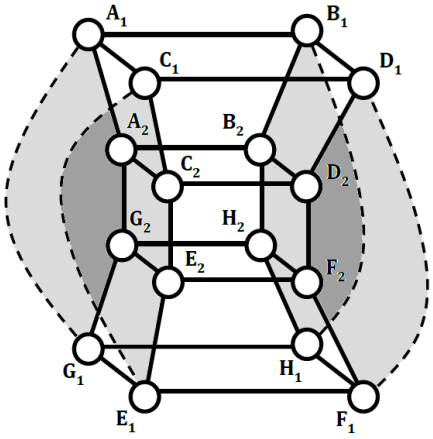
Parallel planes

$A_1A_2B_2B_1$	or	(0, 8, 9, 1)	← plane 0	Set 0
$G_1G_2H_2H_1$	or	(6, 4, 15, 7)	← plane 1	
$C_1C_2D_2D_1$	or	(2, 10, 11, 3)	← plane 2	
$E_1E_2F_2F_1$	or	(4, 12, 13, 5)	← plane 3	



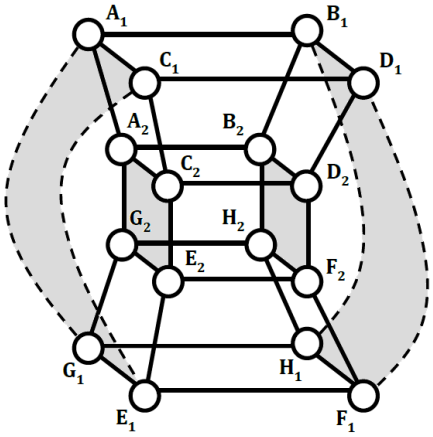
Parallel planes

$C_1C_2A_2A_1$	or	(2, 10, 8, 0)	← plane 4	Set 1
$D_1D_2B_2B_1$	or	(3, 11, 9, 1)	← plane 5	
$E_1E_2G_2G_1$	or	(4, 12, 14, 6)	← plane 6	
$F_1F_2H_2H_1$	or	(5, 13, 15, 7)	← plane 7	



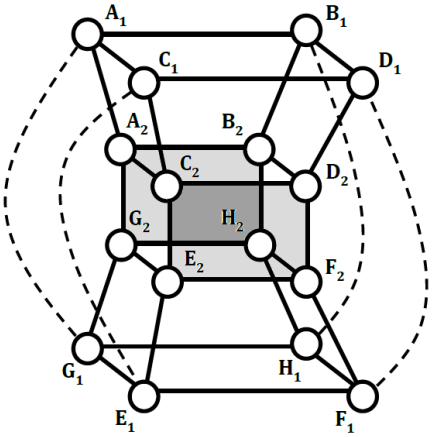
Parallel planes

$A_1G_1G_2A_2$	or	(0, 6, 14, 8)	← plane 8	Set 2
$B_1H_1H_2B_2$	or	(1, 7, 15, 9)	← plane 9	
$C_1E_1E_2C_2$	or	(2, 4, 12, 10)	← plane 10	
$D_1F_1F_2D_2$	or	(3, 5, 13, 11)	← plane 11	



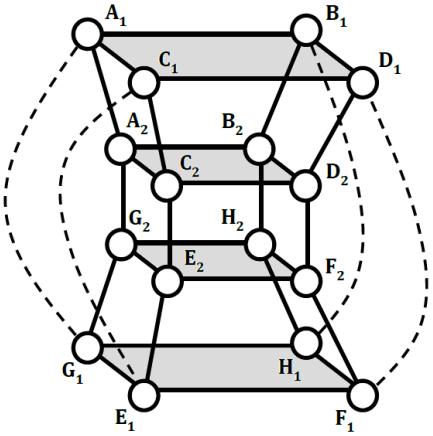
Parallel planes

$D_1B_1H_1F_1$	or	(3, 1, 7, 5)	← plane 12	Set 3
$D_2B_2H_2F_2$	or	(11, 9, 15, 13)	← plane 13	
$C_2A_2G_2E_2$	or	(10, 8, 14, 12)	← plane 14	
$C_1A_1G_1E_1$	or	(2, 0, 6, 4)	← plane 15	



Parallel planes

$C_1E_1F_1D_1$	or	(2, 4, 5, 3)	← plane 16	Set 4
$C_2E_2F_2D_2$	or	(10, 12, 13, 11)	← plane 17	
$A_2G_2H_2B_2$	or	(8, 14, 15, 9)	← plane 18	
$A_1G_1H_1B_1$	or	(0, 6, 7, 1)	← plane 19	



Parallel planes

$A_1B_1D_1C_1$	or	(0, 1, 3, 2)	← plane 20	Set 5
$A_2B_2D_2C_2$	or	(8, 9, 11, 10)	← plane 21	
$G_2H_2F_2E_2$	or	(14, 15, 13, 12)	← plane 22	
$G_1H_1F_1E_1$	or	(6, 7, 5, 4)	← plane 23	

The matrix **M** is a 24X4 matrix (24 rows and 4 columns). Each row of **M** contains 4 integers, representing 4 vertices of a plane (index numbers). Hence the 24 rows represent the 24 planes of the Tesseract. This matrix is constructed by taking

the 6 sets of parallel planes (‘set 1’ to ‘set 5’) i.e., the six 4X4 matrices shown above in order and concatenating them into a single 24X4 matrix.

The figures above also show that each plane of the Tesseract is assigned a plane number. E.g., plane $\mathbf{A_1B_1D_1C_1}$ or **(0, 1, 3, 2)** is assigned the number ‘plane **20**’ (and belongs to ‘set 5’ of parallel planes). In this implementation, a row number of **M** corresponds to the plane number represented by that row. E.g., row 17 contains the elements **(10, 12, 13, 11)** which represent the plane **17**, or row 21 contains the elements **(8, 9, 11, 10)** that represent plane **21**.

M

$A_1A_2B_2B_1$	→	0	8	9	1	plane 0
$G_1G_2H_2H_1$	→	6	14	15	7	plane 1
$C_1C_2D_2D_1$	→	2	10	11	3	plane 2
$E_1E_2F_2F_1$	→	4	12	13	5	plane 3
$C_1C_2A_2A_1$	→	2	10	8	0	plane 4
$D_1D_2B_2B_1$	→	3	11	9	1	plane 5
$E_1E_2G_2G_1$	→	4	12	14	6	plane 6
$F_1F_2H_2H_1$	→	5	13	15	7	plane 7
$A_1G_1G_2A_2$	→	0	6	14	8	plane 8
$B_1H_1H_2B_2$	→	1	7	15	9	plane 9
$C_1E_1E_2C_2$	→	2	4	12	10	plane 10
$D_1F_1F_2D_2$	→	3	5	13	11	plane 11
$D_1B_1H_1F_1$	→	3	1	7	5	plane 12
$D_2B_2H_2F_2$	→	11	9	15	13	plane 13
$C_2A_2G_2E_2$	→	10	8	14	12	plane 14
$C_1A_1G_1E_1$	→	2	0	6	4	plane 15
$C_1E_1F_1D_1$	→	2	4	5	3	plane 16
$C_2E_2F_2D_2$	→	10	12	13	11	plane 17
$A_2G_2H_2B_2$	→	8	14	15	9	plane 18
$A_1G_1H_1B_1$	→	0	6	7	1	plane 19
$A_1B_1D_1C_1$	→	0	1	3	2	plane 20
$A_2B_2D_2C_2$	→	8	9	11	10	plane 21
$G_2H_2F_2E_2$	→	14	15	13	12	plane 22
$G_1H_1F_1E_1$	→	6	7	5	4	plane 23

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

The aforementioned array **T [16]**, representing the set of vertices and storing their values (to be initialized externally with data elements) and the matrix **M [24][4]**, representing the set of planes (constant matrix, storing structural information) are used together to physically implement the Tesseract data model.

This algorithm can be used to encrypt files of any length. The encryption procedure is carried out in 3 stages and each stage uses a different key. The entire plaintext can be considered as a sequence of 12-character blocks. The first stage takes a 12 character block **data [12]** as input and uses it to initialize the vertex set **T [16]** of a Tesseract. This stage uses a text file “otp.txt” as encryption key.

The file “otp.txt” contains a 4X4 matrix $\mathbf{X_1}$, and an integer **N** which later expands to another 4X4 matrix $\mathbf{X_2}$. Each row of $\mathbf{X_1}$ contains a value **INF** (denoted by 9999) at a random position. The remaining 12 positions of the matrix are filled up with some random permutation of the integers lying between 0 and 11, both inclusive. E.g.,

$X_1 =$

10	5	INF	6
2	INF	1	3
7	11	4	INF
8	0	INF	9

The integer **N** refers to the serial number of a parallel plane set, which should be used to initialize the matrix $\mathbf{X_2}$. **N** ranges between 0 and 5 (as there are 6 sets of parallel planes). E.g., let's assume that **N** takes the value 3. So, the matrix $\mathbf{X_2}$ must be filled with the planes (integer quadruplets) belonging to set 3 of parallel planes (refer to matrix **M**), i.e., with the 4X4 sub-matrix of **M**, starting from row **N** x 4 = 3 x 4 = 12.

$X_2 =$

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

After extraction of $\mathbf{X_2}$, another 1X4 vertical matrix **R** is generated at runtime and initialized with 4 random values.

$R =$

RN_0
RN_1
RN_2
RN_3

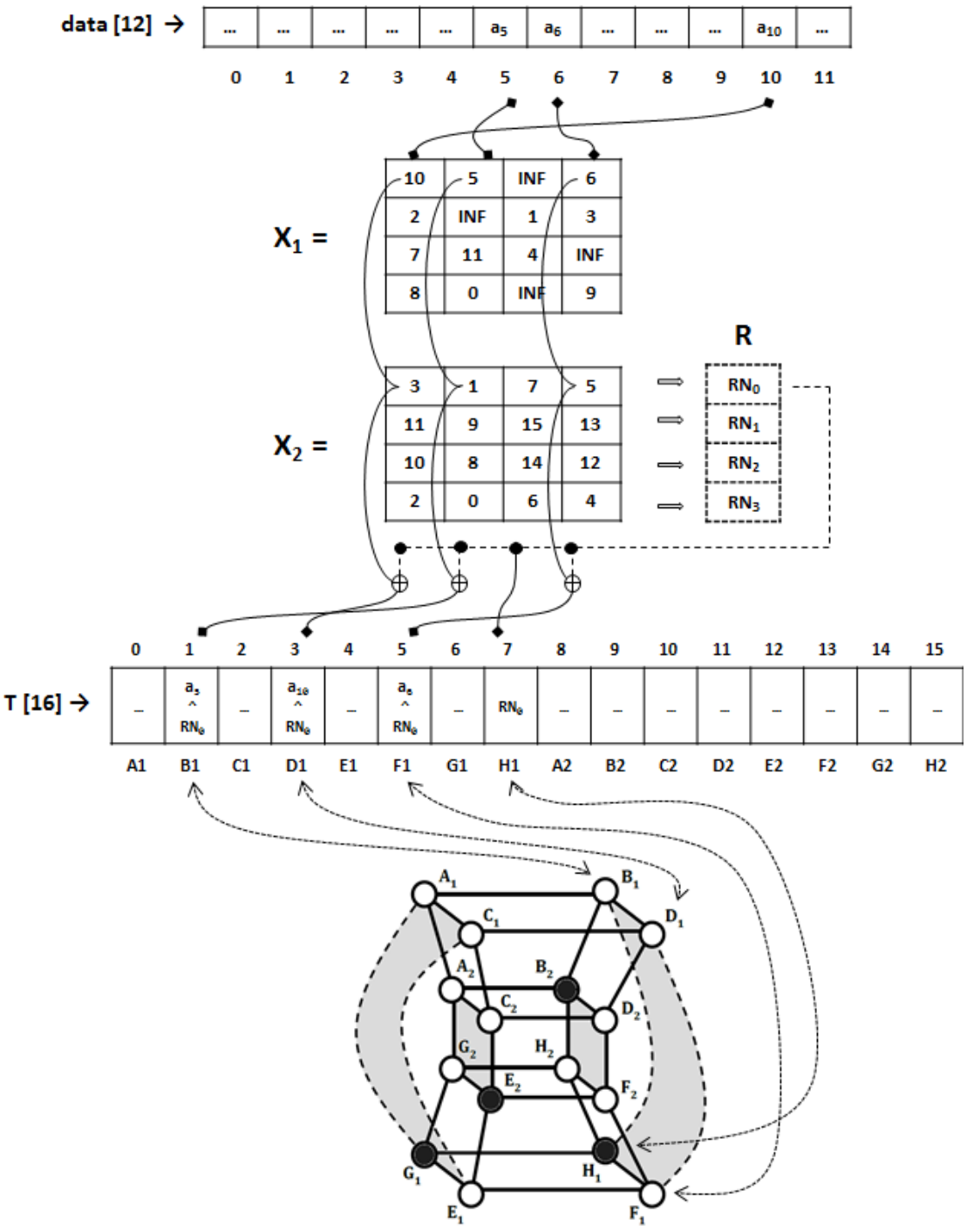
The pseudo code for stage 1 of encryption is given below. The procedure makes use of the key "otp.txt" (to read $\mathbf{X_1}$ and generate $\mathbf{X_2}$) for placing the data elements from **data [12]** into the vertex set **T [16]** to generate the stage 1 ciphertext.

```
Procedure Encrypt_Stage_1 ()
Begin
    For (i = 0 to 3 step +1) do
        For (j = 0 to 3 step +1) do
            If (X1 [i][j] ≠ INF) then
                temp ← data [X1 [i][j]]
                temp ← temp ^ R [i]
                T [X2 [i][j]] ← temp
            Else
                temp ← R [i]
                T [X2 [i][j]] ← temp
            EndIf
        EndFor
    EndFor
EndProcedure
```

Hence, each plane of the selected set ($\mathbf{X_2}$) holds a random number and 3 data elements XORed with this random number in its 4 vertices. For decryption, the stages are traversed in reverse order. I.e., the EA executes stage 1 of encryption first, followed by stage 2 encryption and then stage 3 encryption, whereas the DA executes the stage 3 decryption first, which reverses the operation of stage 3 encryption, then it executes stage 2 decryption which performs the reverse operation of stage 2 encryption, and at last the stage 1 decryption operation is executed which finally produces back the plaintext.

The pseudo code for stage 1 of decryption is given below. This reverses the operations of **`Encrypt_Stage_1 ()`** using the same key "otp.txt" (i.e., $\mathbf{X_1}$ and $\mathbf{X_2}$) to produce the data set **data [12]** from vertex set **T [16]**. A point to be noted here is that the 4 random numbers used in the encryption process are already placed in the vertex set as the 4 redundant elements, wherefrom they can be extracted as well. So, we do not need to know the seed that was used in the EA for random number generation.

```
Procedure Decrypt_Stage_1 ()
Begin
    For (i = 0 to 3 step +1) do
        For (j = 0 to 3 step +1) do
            If (X1 [i][j] = INF) then
                RAND_NO ← T [X2 [i][j]]
            EndIf
        For (j = 0 to 3 step +1) do
            If (X1 [i][j] ≠ INF) then
                temp ← T [X2 [i][j]]
                temp ← temp ^ RAND_NO
                data [X1 [i][j]] ← temp
            EndIf
        EndFor
    EndFor
EndProcedure
```



Stage - 1 of encryption / decryption

Stage 1 encryption is repeated for all subsequent 12-element data blocks to convert them into 16-element vertex sets, till the end of file. Therefore, the output of stage 1 EA is a set containing multiple 16-element subsets of data elements. The decryption process also repeats stage 1 for all the 16-element data sets to produce back corresponding 12-element data blocks which altogether construct the plaintext file.

The second stage of encryption uses a different encryption key that must be externally supplied to the algorithm. A key is a sequence of trivial key-components, where each key-component stands for a specific operation on the data model. The key supplied here is a variable length string consisting of alphabetic characters in the range [**'a' – 'x'**, **'A' – 'X'**]. E.g., "abhEWTRjkOPn" is a valid external key for stage 2 encryption. Here each character (i.e., 'a', 'b', 'h', 'E', 'W', …) represents a trivial key-component. However, the stage 2 encryption algorithm can only recognize and process internal key-components. An internal key-component is represented by a set of 2 integers **{PL_NO, DIR}**, where **PL_NO** ranges from **0** to **23**, and **DIR** is a Boolean, ranging from **0** to **1**. An internal subroutine is used for mapping of external key-components to internal key-components as per the following table:

'a' → {00, 0}	'A' → {00, 1}
'b' → {01, 0}	'B' → {01, 1}
'c' → {02, 0}	'C' → {02, 1}
'd' → {03, 0}	'D' → {03, 1}
'e' → {04, 0}	'E' → {04, 1}
'f' → {05, 0}	'F' → {05, 1}
'g' → {06, 0}	'G' → {06, 1}
'h' → {07, 0}	'H' → {07, 1}
'i' → {08, 0}	'I' → {08, 1}
'j' → {09, 0}	'J' → {09, 1}
'k' → {10, 0}	'K' → {10, 1}
'l' → {11, 0}	'L' → {11, 1}
'm' → {12, 0}	'M' → {12, 1}
'n' → {13, 0}	'N' → {13, 1}
'o' → {14, 0}	'O' → {14, 1}
'p' → {15, 0}	'P' → {15, 1}
'q' → {16, 0}	'Q' → {16, 1}
'r' → {17, 0}	'R' → {17, 1}
's' → {18, 0}	'S' → {18, 1}
't' → {19, 0}	'T' → {19, 1}
'u' → {20, 0}	'U' → {20, 1}
'v' → {21, 0}	'V' → {21, 1}
'w' → {22, 0}	'W' → {22, 1}
'x' → {23, 0}	'X' → {23, 1}

The key is scanned and the key-components are processed one-by-one to perform corresponding operations on the Tesseract. During encryption process, the key is scanned from left to right while the trivial key-components are extracted and supplied to stage 2 EA, but in decryption process, the key is traversed from right to left while the extracted key components are supplied to stage 2 DA.

Stage 2 (EA and DA) is divided into two sub-stages – stage 2.1 and stage 2.2. The stage 2 EA executes stage 2.1 EA followed by stage 2.2 EA for each key-component it encounters, whereas the stage 2 DA does the opposite i.e. it executes stage 2.2 DA followed by stage 2.1 DA for every singular key-component.

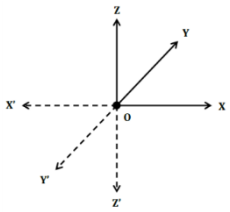
Pseudocode for stage 2 encryption is given below.

```
Procedure Encrypt_Stage_2 ()
Begin
    Key ← user input key string
    For each key_component from left to right
    do
        Encrypt_Stage_2.1 ()
        Encrypt_Stage_2.2 ()
    EndFor
EndProcedure
```

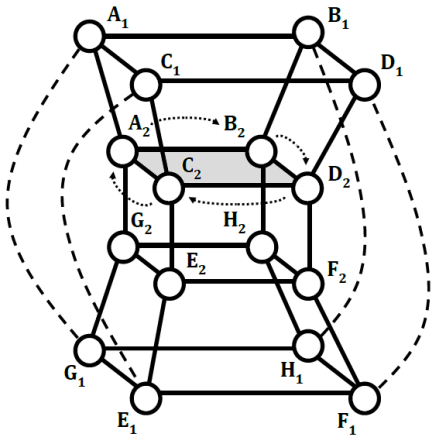
Pseudocode for stage 2 decryption is given below.

```
Procedure Decrypt_Stage_2 ()
Begin
    Key ← user input key string
    For each key_component from right to left
    do
        Decrypt_Stage_2.2 ()
        Decrypt_Stage_2.1 ()
    EndFor
EndProcedure
```

The first sub-stage i.e. stage 2.1 performs a plane-rotation as described by the key-component. Rotations are standardized in compliance with the following axes-orientation.



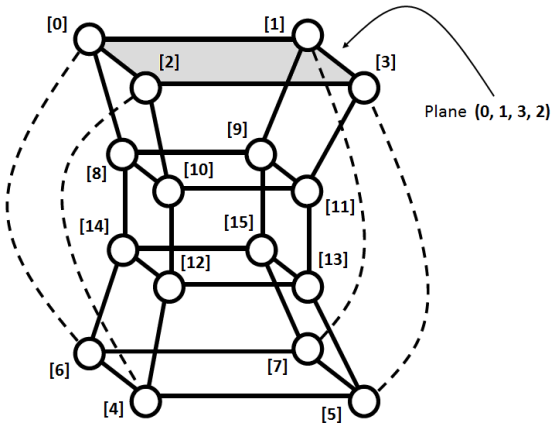
Understanding of axes-orientation or 3-dimentional movement of planes is not particularly important right here. In this implementation, rotation of planes refers to interchanging of values between connected coplanar vertices. E.g., if the plane $\mathbf{A_2B_2D_2C_2}$ be rotated (clockwise), the value stored in $\mathbf{A_2}$ will move to $\mathbf{B_2}$, value in $\mathbf{B_2}$ will move to $\mathbf{D_2}$, value in $\mathbf{D_2}$ will move to $\mathbf{C_2}$ and the value in $\mathbf{C_2}$ will move to $\mathbf{A_2}$.



An internal key-component is denoted by a set of 2 integers **{PL_NO, DIR}**. The number **PL_NO** specifies the serial number of the plane to be rotated. E.g., if **PL_NO = 20**, then plane 20, which is stored at row 20 of **M**, has to be identified first. We have,

M [20] → | 0 | 1 | 3 | 2 |

So, the plane **(0, 1, 3, 2)** is to be rotated.

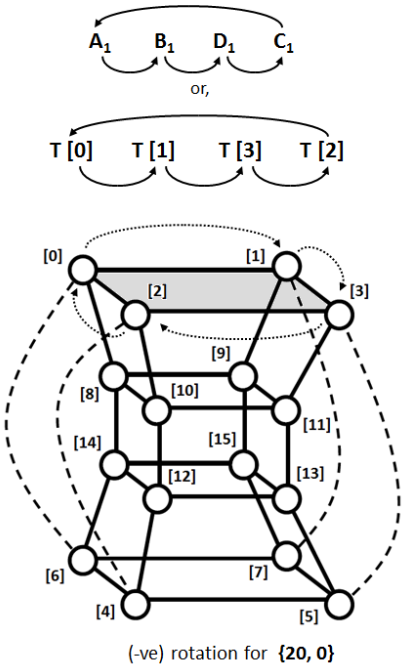


The Boolean **DIR** indicates the direction of rotation. If **DIR = 0**, then a (+ve) or clockwise rotation is performed, meaning, a vertex appearing at any column of the selected row (or plane), passes its value to the vertex appearing at its next column. E.g., in the above example, a (+ve) rotation of the plane **(0, 1, 3, 2)** would move the value from *vertex **0*** to *vertex **1***, *vertex **1*** to *vertex **3***, *vertex **3*** to *vertex **2*** and *vertex **2*** to *vertex **0***.

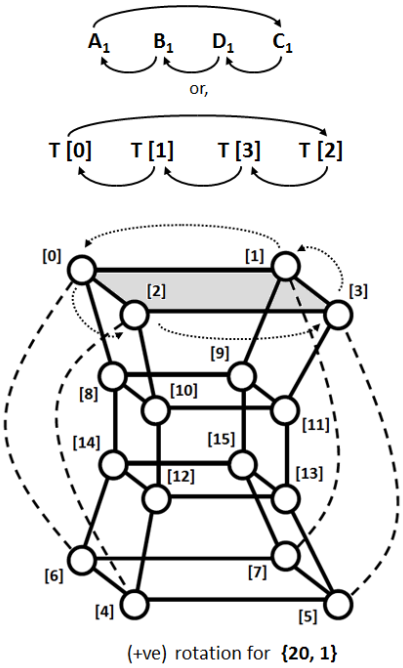
Similarly, If **DIR = 1**, then a (-ve) or anti-clockwise rotation is performed, meaning, a vertex appearing at any column of the selected row (or plane), passes its value to the vertex appearing at its previous column of the same row. E.g., in the above example, a (-ve) rotation of the plane **(0, 1, 3, 2)** would move the value from *vertex **2*** to *vertex **3***, *vertex **3*** to *vertex **1***, *vertex **1*** to *vertex **0*** and *vertex **0*** to *vertex **2***.

M [20] =

0	1	3	2
↑	↑	↑	↑
A_1	B_1	D_1	C_1



(-ve) rotation for **{20, 0}**

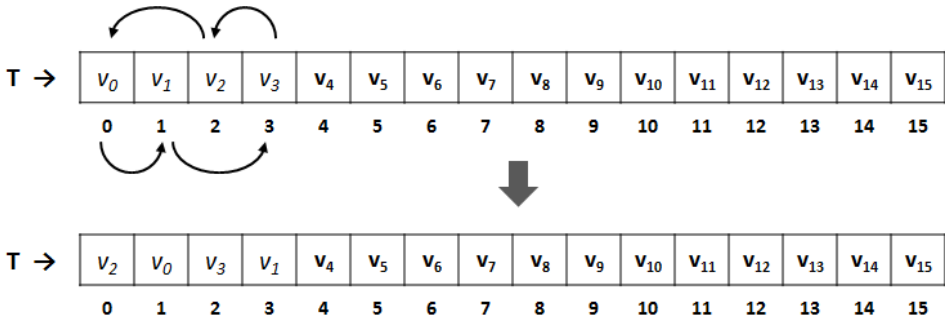


(+ve) rotation for **{20, 1}**

Let's assume that after execution of stage 1 encryption (that initializes the vertex set **T [16]**), array **T** holds the following values.

T →	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Execution of stage 2.1 encryption using external key component **'u'**, that changes to internal key-component **{20, 0}** (used in the above example), will scramble the Tesseract in the following manner.



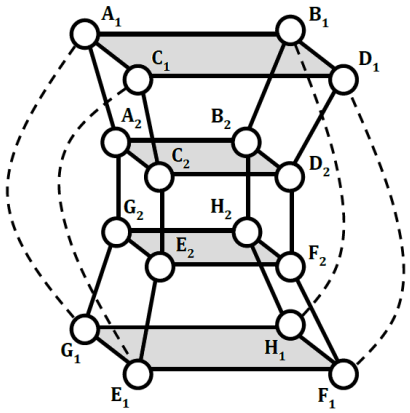
Pseudo code for stage 2.1 encryption is given below. The algorithm receives a single external key-component i.e. an alphabetic character as input, and calls subroutine 'convert ()' to change it into an equivalent internal key-component, which is then interpreted and the corresponding operations are performed.

```
Procedure Encrypt_Stage_2.1 ()
Begin
    {PL_NO, DIR} ← convert (external_key_component)
    temp_array ← M [PL_NO]
    If (DIR = 0)
    then
        circular right shift (T [temp_array [0]],
                              T [temp_array [1]],
                              T [temp_array [2]],
                              T [temp_array [3]])
    Else
        circular left  shift (T [temp_array [0]],
                              T [temp_array [1]],
                              T [temp_array [2]],
                              T [temp_array [3]])

    EndIf
EndProcedure
```

Stage 2.1 DA uses a key-component to reverse the effect of stage 2.1 encryption by performing an almost similar operation with a single alteration – after conversion of external key-component (alphabetic character) to internal key-component (integer pair), the second element of the internal key-component i.e. the Boolean **DIR** is complemented, hence reversing the direction of rotation to nullify the effect of that plane's rotation, performed by stage 2.1 EA in opposite direction.

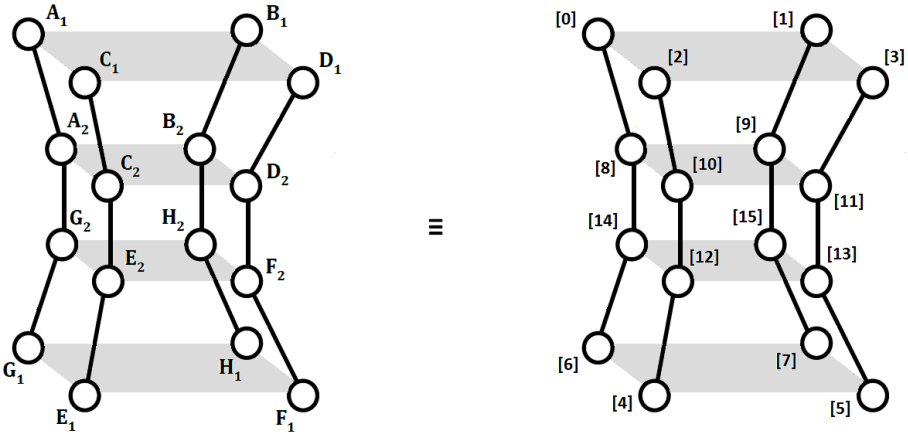
```
Procedure Decrypt_Stage_2.1 ()
Begin
    {PL_NO, DIR} ← passed down from stage 2.2 DA
    temp_array ← M [PL_NO]
    If (DIR = 0)
    then
        circular left  shift (T [temp_array [0]],
                              T [temp_array [1]],
                              T [temp_array [2]],
                              T [temp_array [3]])
    Else
        circular right shift (T [temp_array [0]],
                              T [temp_array [1]],
                              T [temp_array [2]],
                              T [temp_array [3]])

    EndIf
EndProcedure
```

Stage 2.2 of encryption performs grey-code conversion (on data bytes stored in vertices), using the same key-component **{PL_NO, DIR}** used in stage 2.1 EA. First, the parallel plane set, which the plane rotated in stage 2.1 belongs to, is identified. Set number ← $\lfloor$(**PL_NO** / 4)$\rfloor$. In the previous example, the key component **{20, 0}** is processed by stage 2.1 encryption algorithm. Here **PL_NO = 20**, meaning the plane **20** i.e. plane $\mathbf{A_1B_1D_1C_1}$ was rotated, which belongs to Set $\lfloor$(20/4)$\rfloor$ = Set **5** of parallel planes.



Set 5 of parallel planes

The next step is to identify the four aforementioned 4-point lines that connect the parallel planes in the set. In this example, those 4 lines are (**$A_1 – A_2 – G_2 – G_1$**), (**$B_1 – B_2 – H_2 – H_1$**), (**$D_1 – D_2 – F_2 – F_1$**) and (**$C_1 – C_2 – E_2 – E_1$**).



The matrix **M** representing the set of planes is constructed in such a way that for every set of parallel planes (4X4 sub-matrix) belonging to superset **M** (24X4 matrix), collinear vertex quadruplets, wherein each vertex belongs to a different plane (like the 4 quads shown in figure), always reside in the same column. The previous example shows parallel plane set 5, which refers to the 5th 4X4 sub-matrix in **M**, starting from row position 20. The collinear points (**[0] – [8] – [14] – [6]**), (**[1] – [9] – [15] – [7]**), (**[3] – [11] – [13] – [5]**) and (**[2] – [10] – [12] – [4]**) reside in same column.

	19	...	...	...	...
	20	0	1	3	2
M →	21	8	9	11	10
	22	14	15	13	12
	23	6	7	5	4

Set 5

Each of the lines can be considered as sequence of 4 data elements, stored in the vertices (represented by **T**) of the Tesseract. For each line, we read those values in order, convert the sequence into grey-code considering each element as a single data unit, and then put back the modified values into the vertices.

For the previous example, we perform the following operations in stage 2.2 encryption.

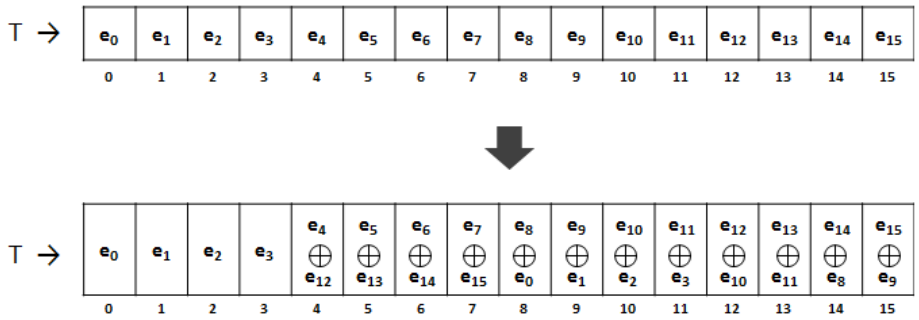
$G_1 \leftarrow G_1 \wedge G_2$ i.e., T [6] ← T [6] ^ T [14]
$G_2 \leftarrow G_2 \wedge A_2$ i.e., T [14] ← T [14] ^ T [8]
$A_2 \leftarrow A_2 \wedge A_1$ i.e., T [8] ← T [8] ^ T [0]
$A_1 \leftarrow A_1$ i.e., T [0] ← T [0]

$H_1 \leftarrow H_1 \wedge H_2$ i.e., T [7] ← T [7] ^ T [15]
$H_2 \leftarrow H_2 \wedge B_2$ i.e., T [15] ← T [15] ^ T [9]
$B_2 \leftarrow B_2 \wedge B_1$ i.e., T [9] ← T [9] ^ T [1]
$B_1 \leftarrow B_1$ i.e., T [1] ← T [1]

$F_1 \leftarrow F_1 \wedge F_2$ i.e., T [5] ← T [5] ^ T [13]
$F_2 \leftarrow F_2 \wedge D_2$ i.e., T [13] ← T [13] ^ T [11]
$D_2 \leftarrow D_2 \wedge D_1$ i.e., T [11] ← T [11] ^ T [3]
$D_1 \leftarrow D_1$ i.e., T [3] ← T [3]

$E_1 \leftarrow E_1 \wedge E_2$ i.e., T [4] ← T [4] ^ T [12]
$E_2 \leftarrow E_2 \wedge C_2$ i.e., T [12] ← T [12] ^ T [10]
$C_2 \leftarrow C_2 \wedge C_1$ i.e., T [10] ← T [10] ^ T [2]
$C_1 \leftarrow C_1$ i.e., T [2] ← T [2]

Suppose the vertex set **T** takes the following values after stage 2.1 encryption. The stage 2.2 encryption, subsequently carried out using the same key-component **{20, 0}** (second element **DIR** is immaterial here), will alter its values as shown below.



Pseudo code for stage 2.2 encryption is given below.

```
Procedure Encrypt_Stage_2.2 ()
Begin
    {PL_NO, DIR} ← passed down from stage 2.1 EA
    Set_No ← floor (PL_NO / 4)
    temp_4X4_mtrx ← {M [Set_No * 4],
                     M [Set_No * 4 + 1],
                     M [Set_No * 4 + 2],
                     M [Set_No * 4 + 3]}
    For (i = 0 to 3 step +1)
    do
        convert_to_grey (T [temp_4X4_mtrx [0][i]],
                         T [temp_4X4_mtrx [1][i]],
                         T [temp_4X4_mtrx [2][i]],
                         T [temp_4X4_mtrx [3][i]])
    EndFor
EndProcedure
```

Decryption of stage 2.2 involves performing the reverse operation of grey conversion on each column. For the previous example, stage 2.2 decryption procedure performs the following operations on **T**.

$A_1 \leftarrow A_1$ i.e., T [0] ← T [0]
$A_2 \leftarrow A_2 \wedge A_1$ i.e., T [8] ← T [8] ^ T [0]
$G_2 \leftarrow G_2 \wedge A_2$ i.e., T [14] ← T [14] ^ T [8]
$G_1 \leftarrow G_1 \wedge G_2$ i.e., T [6] ← T [6] ^ T [14]

$B_1 \leftarrow B_1$ i.e., T [1] ← T [1]
$B_2 \leftarrow B_2 \wedge B_1$ i.e., T [9] ← T [9] ^ T [1]
$H_2 \leftarrow H_2 \wedge B_2$ i.e., T [15] ← T [15] ^ T [9]
$H_1 \leftarrow H_1 \wedge H_2$ i.e., T [7] ← T [7] ^ T [15]

$D_1 \leftarrow D_1$ i.e., T [3] ← T [3]
$D_2 \leftarrow D_2 \wedge D_1$ i.e., T [11] ← T [11] ^ T [3]
$F_2 \leftarrow F_2 \wedge D_2$ i.e., T [13] ← T [13] ^ T [11]
$F_1 \leftarrow F_1 \wedge F_2$ i.e., T [5] ← T [5] ^ T [13]

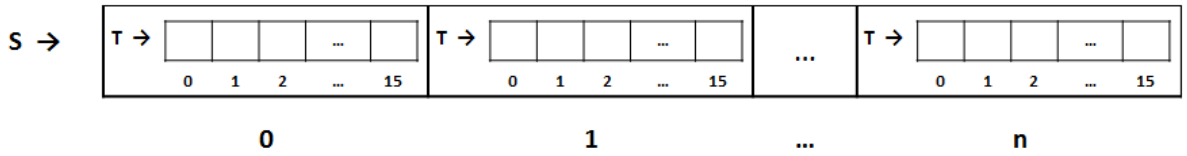
$C_1 \leftarrow C_1$ i.e., T [2] ← T [2]
$C_2 \leftarrow C_2 \wedge C_1$ i.e., T [10] ← T [10] ^ T [2]
$E_2 \leftarrow E_2 \wedge C_2$ i.e., T [12] ← T [12] ^ T [10]
$E_1 \leftarrow E_1 \wedge E_2$ i.e., T [4] ← T [4] ^ T [12]

Pseudo code for stage 2.2 decryption is given below.

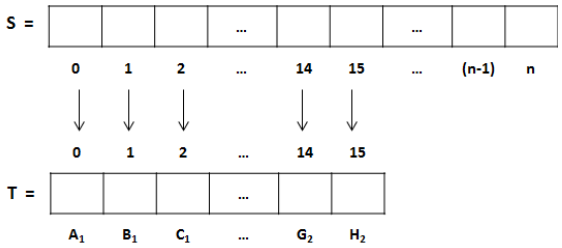
```
Procedure Decrypt_Stage_2.2 ()
Begin
    {PL_NO, DIR} ← convert (external_key_component)
    Set_No ← floor (PL_NO / 4)
    temp_4X4_mtrx ← {M [Set_No * 4],
                     M [Set_No * 4 + 1],
                     M [Set_No * 4 + 2],
                     M [Set_No * 4 + 3]}
    For (i = 0 to 3 step +1)
    do
        undo_grey_conversion (T [temp_4X4_mtrx [0][i]],
                              T [temp_4X4_mtrx [1][i]],
                              T [temp_4X4_mtrx [2][i]],
                              T [temp_4X4_mtrx [3][i]])
    EndFor
EndProcedure
```

Stage 2 EA is repeated using same encryption key for every 16-element vertex set i.e. on every distinct Tesseract constructed in the previous stage (stage 1 EA). Therefore, the output of stage 2 EA is the same set it receives from stage 1, comprising multiple Tesseracts (16-element subsets), with each one having undergone specified operations. Stage 2 DA is also executed using a single decryption key (same as the stage 2 encryption key) on every distinct Tesseract returned from Stage 3 DA.

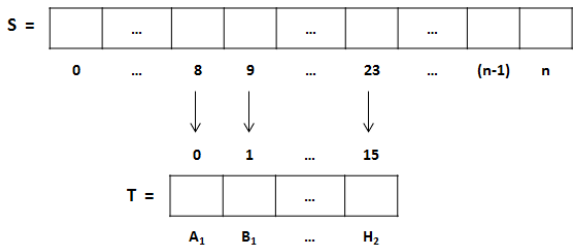
For stage 3 encryption, we take the array of Tesseracts, say **S**, formed in stage 2. Each element of the array **S** is again a 16-element array representing vertex set of a Tesseract.



In this stage, we construct Tesseracts from array **S**, using the aforementioned convention for mapping of Tesseract vertices with array indices. E.g., the first 16 elements of **S** construct the first Tesseract.



The second Tesseract starts from 8th offset of **S**, the third one starts from 16th offset, the fourth one from 24th offset and so on. Hence, every mth Tesseract starts from ((m - 1) * 8)th offset of **S** and contains sixteen consecutive elements. The figure below shows construction of the second Tesseract from **S**.



Unlike stage 1 and stage 2, which constructs and operates on multiple distinct Tesseracts respectively (composed of disjoint vertex-sets), stage 3 constructs a series of Tesseracts where any two consecutive Tesseracts have 8 vertices in common. So when we operate on a Tesseract, other two situated at both sides of it are affected as well.

The stage 3 EA uses a user-input variable length alphanumeric key-string where key-components i.e. the characters in the string lie in the range **['a' – 'x', 'A' – 'X', '1' – '2']**. At first, the first Tesseract, starting at the 0^{th} offset (**S [0]** to **S [15]**), say **T**, is extracted from **S**. The encryption key is scanned from left to right, and for the alphabetic characters, the exact same shuffling operations as in stage 2.1 are performed on **T** (only instead of individual data bytes, 16-element data blocks are interchanged between vertices).

When a numeric character is encountered,

i. Firstly, the values in **T** are put back into the corresponding positions of **S**.
ii. Now, if the character is '**1**' then the right hand side Tesseract of the previous one is extracted in **T** from **S**, and if the character is '**2**' then the left hand side Tesseract of the previous one is extracted in **T** from **S**.

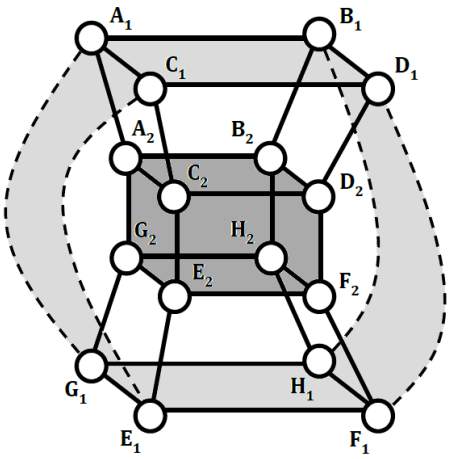
Let's take a stage 3 encryption key "cb1a2f" for example. At the beginning of stage 3 EA execution, **T** holds the vertex set of the first Tesseract (i.e., the values from **S [0]** to **S [15]**).

- The key-components 'c' and 'b' scramble the values in **T** in the same pattern as described in stage 2.1 of EA.
- When the key-component '1' is encountered, first we move the values of **T** in **S [0]** to **S [15]**. Then we extract the right hand side Tesseract (i.e., the second Tesseract) in **T**, meaning, the values in **S [8]** to **S [23]** are copied to **T**.
- The key-component 'a' scrambles **T** accordingly.
- When the key-component '2' is scanned, first, the values in **T** are moved in **S [8]** to **S [23]**. Then the left hand side Tesseract of the previous one (i.e., the first Tesseract, beginning at **S [0]** and ending at **S [15])** is copied to **T**.
- Key-component 'f' scrambles **T** accordingly.
- At the end of operation, elements of **T** are put back in the locations **S [0]** to **S [15]** wherefrom they were lastly extracted (for the 5^{th} key-component '2').

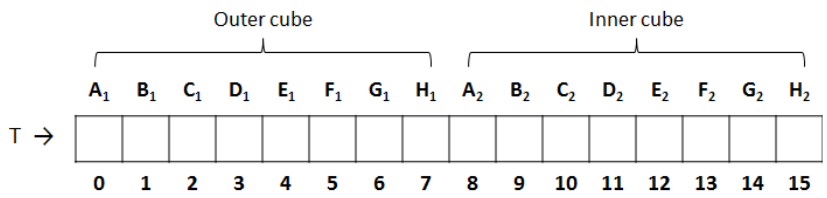
After execution of stage 3 EA, contents of **S** represent the ciphertext, which is then written block by block in the output file. Stage 3 DA reads ciphertext from a file and copies it to **S**. Then it reverses the stage 3 encryption operation using the same key used in stage 3 EA, by traversing it in reverse order while performing opposite operations for numeric key-components '**1**' & '**2**', and the exact same shuffles as of stage 2.1 DA for the alphabetic key-components.

The overlapping of consecutive Tesseracts constructed at this stage geometrically signifies operating on a multilevel structure of Tesseract. Here the fourth structural unit of a Tesseract is utilized.

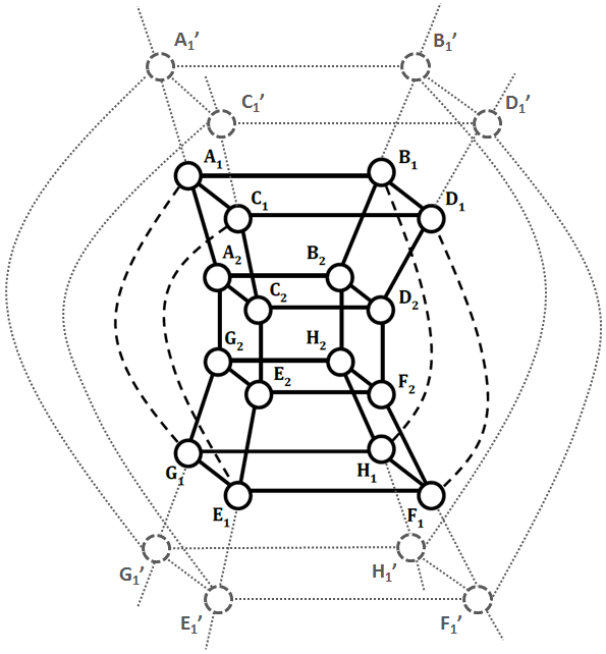
- Cube – a Tesseract consists of 8 cubes, each enclosed with 8 vertices. For this document, we only consider the outermost cube (consisting of $\mathbf{A_1}$, $\mathbf{B_1}$, $\mathbf{C_1}$, $\mathbf{D_1}$, $\mathbf{E_1}$, $\mathbf{F_1}$, $\mathbf{G_1}$, $\mathbf{H_1}$) and the innermost cube (consisting of $\mathbf{A_2}$, $\mathbf{B_2}$, $\mathbf{C_2}$, $\mathbf{D_2}$, $\mathbf{E_2}$, $\mathbf{F_2}$, $\mathbf{G_2}$, $\mathbf{H_2}$) of a Tesseract.



Now for the array **T**, that represents the set of vertices of a Tesseract, we see as per the mapping convention we used earlier, all of the 8 vertices that belong to the outer cube reside in the first 8-element subset of **T**, and the rest of 8 vertices that belong to the inner cube reside in the trailing 8-element subset of **T**.



From this we can say that the innermost cube of any stage 3 Tesseract becomes the outermost cube of its right hand side Tesseract, or alternatively, the outermost cube of any Tesseract is same as the innermost cube of its left hand side Tesseract. Hence, any Tesseract starting from m^{th} index of **S** (where 'm' clearly is a multiple of 8) geometrically holds another Tesseract starting from $(m+8)^{th}$ index in its innermost cube, which again contains another Tesseract starting from $(m+16)^{th}$ index inside its innermost cube, and so on. This way we can achieve a multilevel Tesseract structure, where any two Tesseracts belonging to consecutive levels can share data through the common inner-outer cube.



Pseudocode for stage 3 encryption is given below.

```
Procedure Encrypt_Stage_3 ()
Begin
    offset ← 0
    T ← S [offset] to S [offset + 15]
    For each key_component from left to right
    do
        If (key_component is alphabet)
        then
            {PL_NO, DIR} ← convert (key_component)
            temp_array ← M [PL_NO]
            If (DIR = 0)
            then
                circular right shift (T [temp_array [0]],
                                      T [temp_array [1]],
                                      T [temp_array [2]],
                                      T [temp_array [3]])
            Else
                circular left  shift (T [temp_array [0]],
                                      T [temp_array [1]],
                                      T [temp_array [2]],
                                      T [temp_array [3]])
            EndIf
        Else If (key_component = '1')
        then
            S [offset] to S [offset + 15] ← T
            offset ← offset + 8
            T ← S [offset] to S [offset + 15]
        Else If (key_component = '2')
        then
            S [offset] to S [offset + 15] ← T
            offset ← offset - 8
            T ← S [offset] to S [offset + 15]
        EndIf
    EndFor
EndProcedure
```

Pseudocode for stage 3 decryption is given below.

```
Procedure Encrypt_Stage_3 ()
Begin
    offset ← 0
    For each key_component from left to right
    do
        If (key_component = ‘1’)
        then
            offset ← offset + 8
        Else If (key_component = ‘2’)
        then
            offset ← offset – 8
        EndIf
    T ← S [offset] to S [offset + 15]
    For each key_component from right to left
    do
        If (key_component is alphabet)
        then
            {PL_NO, DIR} ← convert (key_component)
            temp_array ← M [PL_NO]
            If (DIR = 0)
            then
                circular left  shift (T [temp_array [0]],
                                      T [temp_array [1]],
                                      T [temp_array [2]],
                                      T [temp_array [3]])
            Else
                circular right shift (T [temp_array [0]],
                                      T [temp_array [1]],
                                      T [temp_array [2]],
                                      T [temp_array [3]])
            EndIf
        Else If (key_component = ‘1’)
        then
            S [offset] to S [offset + 15] ← T
            offset ← offset - 8
            T ← S [offset] to S [offset + 15]
        Else If (key_component = ‘2’)
        then
            S [offset] to S [offset + 15] ← T
            offset ← offset + 8
            T ← S [offset] to S [offset + 15]
        EndIf
    EndFor
EndProcedure
```

Source Code :

```
/*
 |----------|-------------------------------------------------------------|
 | Encipher | <prog. name> <E> <inp. file> <outp. file> <key 2> <key 3>  |
 |----------|-------------------------------------------------------------|
 | Decipher | <prog. name> <D> <inp. file> <outp. file> <key 2> <key 3>  |
 |----------|-------------------------------------------------------------|
*/

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

/*=============================================
       THE 24 PLANES AS VERTEX QUADRUPLETS
=============================================*/

short planes [24][4] =
    {
     0,  8,  9,  1,
     6, 14, 15,  7,
     2, 10, 11,  3,
     4, 12, 13,  5,

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

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

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

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

     0,  1,  3,  2,
     8,  9, 11, 10,
    14, 15, 13, 12,
     6,  7,  5,  4
    };

/*=============================================
      FIRST LEVEL INITIALIZATION USING OTP
=============================================*/

#define  OTP  "otp.txt"

short prll_quad_mtr_no,
      mpp_actual_to_tssr [4][4];

void init_otp ();
void get_tss  (char [12], char [16]);
void put_tss  (char [16], char [12]);

/*=============================================
            ROTATIONS OF THE PLANES
=============================================*/

short rot_id [] =
    {
    0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230,
    1, 11, 21, 31, 41, 51, 61, 71, 81, 91, 101, 111, 121, 131, 141, 151, 161, 171, 181, 191, 201, 211, 221, 231
    };

void rotate   (char [16], short, short);
void bin2gry  (char [16], short [4][4]);
void gry2bin  (char [16], short [4][4]);
void e_scrmbl (char [16], char*);
```

```
void d_scrmbl (char [16], char*);

/*============================================
          SECOND LEVEL INITIALIZATION
============================================*/

void get_blk_tss (char  [], char* [], long);
void put_blk_tss (char* [], char  [], long);

/*============================================
            ROTATIONS OF THE BLOCKS
============================================*/

void rotblk       (char* [16], short, short);
void e_blk_scrmbl (char* [],   long,  char*);
void d_blk_scrmbl (char* [],   long,  char*);

/*============================================
     THE 'main ()' FUNCTION AS DRIVER MODULE
============================================*/

int main (int argc, char *argv [])
    {
    FILE *f1, *f2;
    char *x, *y, **z;
    long  sze1, sze2, sze3,
          i, tmp;

    if (argc ^ 6    ||
        argv [1][1] ||
        !(f1 = fopen (argv [2], "rb")) ||
        !(f2 = fopen (argv [3], "wb")))
            return 0;

    init_otp ();

    if (argv [1][0] == 'E' || argv [1][0] == 'e')
        {
        fseek (f1, 0, SEEK_END);
        tmp = ftell (f1);
        rewind (f1);

        sze1 = ceil (tmp / 12.0) * 12;
        x = (char*) malloc (sze1 * sizeof (char));
        sze2 = sze1 / 12 * 16;
        y = (char*) malloc (sze2 * sizeof (char));
        sze3 = sze2 / 16;
        z = (char**) malloc (sze3 * sizeof (char*));

        for (i = 0 ; i < tmp ; i++)
            fread (x + i, 1, 1, f1);
        while (i < sze1) x [i++] = ' ';

        srand (time (NULL));
        for (i = sze1 ; i ; i -= 12)
            get_tss  (x, y),
            e_scrmbl (y, argv [4]),
            x += 12, y += 16;
        y -= sze2;

        if (sze3 >= 16)
            get_blk_tss  (y, z, sze3),
            e_blk_scrmbl (z, sze3, argv [5]),
            put_blk_tss  (z, y, sze3);

        for (i = 0 ; i < sze2 ; i++)
            fwrite (y + i, 1, 1, f2);
        }
    else if (argv [1][0] == 'D' || argv [1][0] == 'd')
        {
        fseek (f1, 0, SEEK_END);
        tmp = ftell (f1);
        rewind (f1);

        sze1 = ceil (tmp / 16.0) * 16;
        y = (char*) malloc (sze1 * sizeof (char));
        sze3 = sze1 / 16;
        z = (char**) malloc (sze3 * sizeof (char*));
        sze2 = sze1 / 16 * 12;
```

```
        x = (char*) malloc (sze2 * sizeof (char));

        for (i = 0 ; i < tmp ; i++)
            fread (y + i, 1, 1, f1);
        while (i < sze1) y [i++] = ' ';

        if (sze3 >= 16)
            get_blk_tss  (y, z, sze3),
            d_blk_scrmbl (z, sze3, argv [5]),
            put_blk_tss  (z, y, sze3);

        for (i = sze1 ; i ; i -= 16)
            d_scrmbl (y, argv [4]),
            put_tss  (y, x),
            y += 16, x += 12;
        x -= sze2;

        for (i = 0 ; i < sze2 ; i++)
            fwrite (x + i, 1, 1, f2);
        }

    return 0;
    }

/*============================================
      DEFINITIONS OF THE UTILITY FUNCTIONS
============================================*/

void init_otp ()
    {
    FILE *fp = fopen (OTP, "rt");
    int  i;

    fscanf (fp, "%hi", &prll_quad_mtr_no);
    for (i = 0 ; i < 16 ; i++)
        fscanf (fp, "%hi", mpp_actual_to_tssr [i / 4] + (i % 4));
    }

void get_tss (char inp [12], char a [16])
    {
    short *x = planes [4 * prll_quad_mtr_no],
          *y = mpp_actual_to_tssr [0],
           i, j;
    char tmp;

    for (i = 0 ; i < 4 ; i++)
    for (tmp = rand () % 256,
         j = 0 ; j < 4 ; j++)
            a [*x] = *y ^ 9999 ? tmp ^ inp [*y] : tmp,
            x++, y++;
    }

void put_tss (char a [16], char out [12])
    {
    short *x = planes [4 * prll_quad_mtr_no],
          *y = mpp_actual_to_tssr [0],
           i, j;
    char tmp;

    for (i = 0 ; i < 4 ; i++)
        {
        for (j = 0 ; y [j] ^ 9999 ; j++);
        for (tmp = a [x [j]], j = 0 ; j < 4 ; j++)
            *y ^ 9999 && (out [*y] = tmp ^ a [*x]),
            x++, y++;
        }
    }

void rotate (char a [16], short pln_no, short dir)   // dir : {0, 1}
    {
    short *T = planes [pln_no] + 3 * !dir,
           d = -dir | 1;                             // dir : 0 -> (+1) Right , 1 -> (-1) Left

    char C         = a [T [0]];
    a [T [0]]      = a [T [-d]];
    a [T [-d]]     = a [T [-d * 2]];
    a [T [-d * 2]] = a [T [-d * 3]];
    a [T [-d * 3]] = C;
    }
```

```
void bin2gry (char a [16], short pl [4][4])
    {
    int i;

    for (i = 0 ; i < 4 ; i++)
        a [pl [3][i]] ^= a [pl [2][i]],
        a [pl [2][i]] ^= a [pl [1][i]],
        a [pl [1][i]] ^= a [pl [0][i]];
    }

void gry2bin (char a [16], short pl [4][4])
    {
    int i;

    for (i = 0 ; i < 4 ; i++)
        a [pl [1][i]] ^= a [pl [0][i]],
        a [pl [2][i]] ^= a [pl [1][i]],
        a [pl [3][i]] ^= a [pl [2][i]];
    }

void e_scrmbl (char a [16], char *sq)
    {
    char c;

    while (c = *sq++)
        if (c >= 'a' && c <= 'x')
            rotate  (a, rot_id [c - 'a'] / 10,
                        rot_id [c - 'a'] % 10),
            bin2gry (a, (short (*)[4]) (planes + rot_id [c - 'a'] / 40 * 4));
        else if (c >= 'A' && c <= 'X')
            rotate  (a, rot_id [c - 'A' + 24] / 10,
                        rot_id [c - 'A' + 24] % 10),
            bin2gry (a, (short (*)[4]) (planes + rot_id [c - 'A' + 24] / 40 * 4));
    }

void d_scrmbl (char a [16], char *sq)
    {
    char  c;
    short i;

    for (i = 0 ; *sq ; i++, sq++);

    while (i-- && (c = *--sq))
        if (c >= 'a' && c <= 'x')
            gry2bin (a, (short (*)[4]) (planes + rot_id [c - 'a'] / 40 * 4)),
            rotate  (a, rot_id [c - 'a'] / 10,
                       !(rot_id [c - 'a'] % 10));
        else if (c >= 'A' && c <= 'X')
            gry2bin (a, (short (*)[4]) (planes + rot_id [c - 'A' + 24] / 40 * 4)),
            rotate  (a, rot_id [c - 'A' + 24] / 10,
                       !(rot_id [c - 'A' + 24] % 10));
    }

void get_blk_tss (char a [], char *z [], long len)
    {
    long i, j;

    for (i = j = 0 ; j < len ; i += 16, j++)
        z [j] = a + i;
    }

void put_blk_tss (char *z [], char a [], long len)
    {
    int  i, j;
    char *tmp = (char*) malloc (len * 16);

    for (i = 0 ; i < len ; i++)
    for (j = 0 ; j < 16 ; j++)
        tmp [i * 16 + j] = z [i][j];

    for (i = len * 16 - 1 ; i >= 0 ; i--)
        a [i] = tmp [i];
    free (tmp);
    }

void rotblk (char *z [16], short pln_no, short dir)   // dir : {0, 1}
    {
    short *T = planes [pln_no] + 3 * !dir,
          d = -dir | 1;                               // dir : 0 -> (+1) Right , 1 -> (-1) Left
```

```
    char *P        = z [T [0]];
    z [T [0]]      = z [T [-d]];
    z [T [-d]]     = z [T [-d * 2]];
    z [T [-d * 2]] = z [T [-d * 3]];
    z [T [-d * 3]] = P;
    }

void e_blk_scrmbl (char *z [], long len, char *sq)
    {
    long L = 0;
    char c;

    while (c = *sq++)
        if (c >= 'a' && c <= 'x')
            rotblk  (z + L, rot_id [c - 'a'] / 10,
                            rot_id [c - 'a'] % 10);
        else if (c >= 'A' && c <= 'X')
            rotblk  (z + L, rot_id [c - 'A' + 24] / 10,
                            rot_id [c - 'A' + 24] % 10);
        else if (c == '1') L + 23 < len && (L += 8);
        else if (c == '2') L > 7 && (L -= 8);
        else if (c == '3') ;
    }

void d_blk_scrmbl (char *z [], long len, char *sq)
    {
    long  L = 0;
    char  c;
    short i;

    for (i = 0 ; *sq ; i++, sq++)
        if      (*sq == '1') L + 23 < len ? (L += 8) : (*sq = '3');
        else if (*sq == '2') L > 7 ? (L -= 8) : (*sq = '3');

    while (i-- && (c = *--sq))
        if (c >= 'a' && c <= 'x')
            rotblk  (z + L, rot_id [c - 'a'] / 10,
                           !(rot_id [c - 'a'] % 10));
        else if (c >= 'A' && c <= 'X')
            rotblk  (z + L, rot_id [c - 'A' + 24] / 10,
                           !(rot_id [c - 'A' + 24] % 10));
        else if (c == '2') L += 8;
        else if (c == '1') L -= 8;
        else if (c == '3') ;
    }
```