**Fundamentos de Programación (9 de septiembre de 2016)**

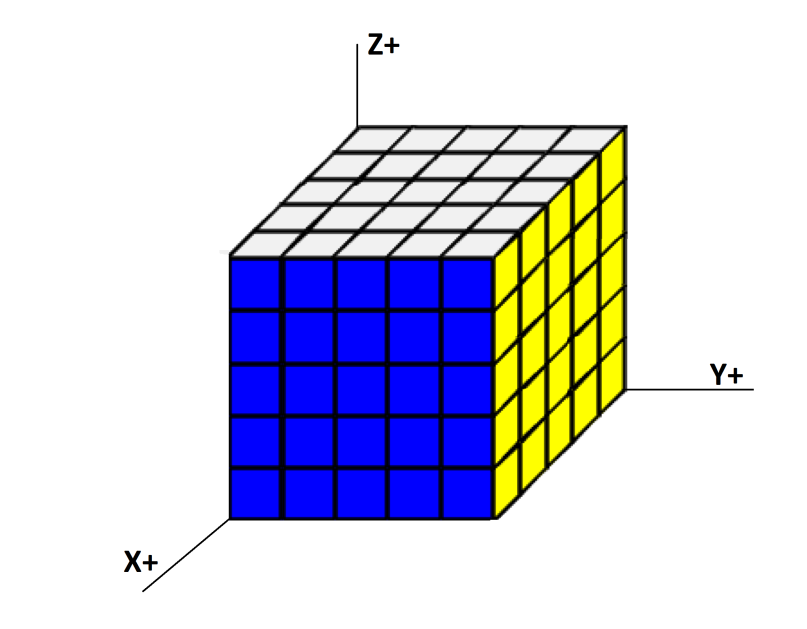
**(Grados en Ingeniería Mecánica, Eléctrica, Electrónica Industrial y Química Industrial)**

***Ejercicio:*** construir un programa en ***C*** lo más modular posible (atendiendo a los criterios de modularidad) que se ajuste a la especificación dada y documentar el diseño preliminar con la definición de las nuevas tipologías de datos, el diagrama de módulos (estructura del programa)y las interfaces de los módulos, y el diseño detallado con las definiciones de los respectivos sub-programas.

**Ejercicio:** Construir un programa para simular los movimientos de giro en la resolución de un cubo o puzzle de ***Rubik*** de longitud de lado ***n*** unidades. Un cubo de ***Rubik*** podemos considerarlo como un cubo grande compuesto en principio por ***n3*** cubos unitarios ó piezas, que en el estado ordenado cada cara del cubo es de un color. Existen 4 tipos de piezas según su posición relativa en el cubo: vértices (***8***), aristas (***12\*(n-2)***), centros (***6\*(n-2)2***) e interiores (***(n-2)3***). Los vértices y aristas si tienen realmente forma de cubo, con una particular pestaña de sujeción. Los centros son en realidad medios cubos y están sujetos mediante un eje a un sistema de crucetas centrales (mecanismo interno que permite las posibilidades de giro de las caras). Los interiores son los que conforman dicho mecanismo interno y no son visibles.

En el ejemplo se muestra un cubo de ***Rubik*** de lado ***5***, con un sistema de referencia inicial para identificar sus caras, junto con la nomenclatura de las mismas y su configuración de colores en el estado ordenado inicial:

|  |  |  |  |
| --- | --- | --- | --- |
| **Cara** | **Vector normal** | **Nomenclatura** | **Color inicial** |
| 0 | X+ | Frontal | A: azul |
| 1 | Y+ | Derecha | M: amarillo |
| 2 | Z+ | Arriba | B: blanco |
| 3 | X- | Trasera | N: negro |
| 4 | Y- | Izquierda | R: rojo |
| 5 | Z- | Abajo | V: verde |



Capa n-1

...

Eje Z+ ...

...

Capa 1

Capa 0

Capa 0

Capa 1

...

...

...

Capa n-1

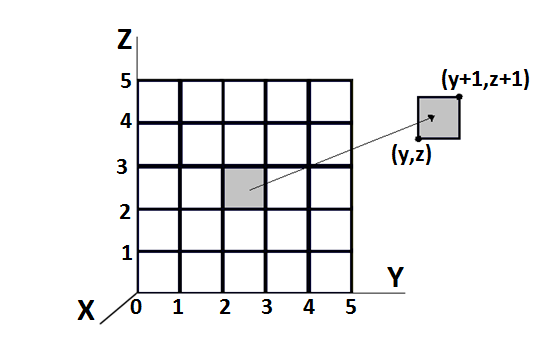
Eje X+ Capa 0

Capa1

...

Capa n-1

Eje Y+



Un detalle importante es que las piezas unitarias sólo tienen pegatinas de colores en las caras visibles inicialmente lo que permite simplificar la información a representar. Para identificar la posición de una cara visible de una pieza dentro del plano a la que pertenece, se utilizarán las coordenadas de intersección de línea de cuadrícula de una de sus esquinas. Ejemplo: ***n=5***, Plano ***X+*** (ó ***X-***):

Nota: para los otros planos, aplicar simetría mediante el intercambio de los ejes en orden: X🡪Y🡪Z🡪X (0🡪1🡪2🡪0)

El movimiento básico del cubo de ***Rubik*** es el giro de ***90º*** alrededor de un eje paralelo a uno de los ejes cartesianos y que pasa por el centro del cubo. Con respecto a dicho eje de rotación, podemos considerar el cubo dividido en ***n*** niveles ó capas perpendiculares al eje de rotación. Un movimiento del cubo vendría por tanto descrito por los siguientes parámetros:

* Eje de rotación: éste atraviesa el centro del cubo y solo se admiten tres posibilidades: 0: paralelo a X+, 1: paralelo a Y+, 2: paralelo a Z+.
* Capa o nivel: nº entero en el intervalo ***[0,n-1]*** que representa al plano de cubos unitarios perpendicular al eje de rotación y numerado según el sistema de referencia inicial.
* Ángulo de giro: debe de ser un múltiplo entero de ***90º*** (positivo en sentido anti-horario con respecto al eje de rotación). Nótese que un giro de ***180º*** equivale a dos giros consecutivos de ***90º***, y que un giro de ***270º*** (ó ***-90º***) es igual a tres giros consecutivos de ***90º***.

Ejemplo: ***n=5***, **giro 90º, capa 1, eje X+**

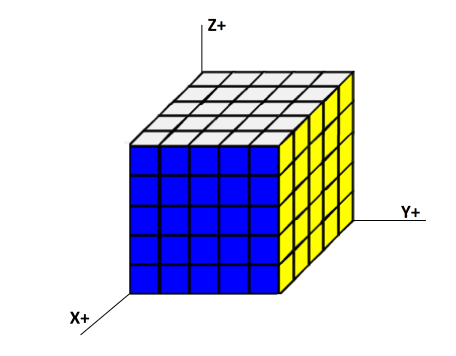
|  |  |  |  |
| --- | --- | --- | --- |
| **Estado inicial** | **Capa 1 eje X+** | **Giro 90º capa 1** | **Estado final** |
|  |  |  |  |

Nota: para giros alrededor de otro eje, aplicar simetría mediante el intercambio de los ejes en orden: X🡪Y🡪Z🡪X (0🡪1🡪2🡪0)

El programa solicitará inicialmente por teclado la longitud ***n*** del lado del cubo de ***Rubik*** en piezas (mínimo ***1*** y máximo ***10***). Seguidamente inicializará los colores de las caras visibles de las piezas con sus correspondientes valores en su estado ordenado inicial, e imprimirá dicho estado en pantalla simulando una proyección paralela axonométrica de las caras ocultas y visibles del cubo, según el formato indicado en el siguiente ejemplo:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Caras ocultas** | |  | **Caras visibles** | |
| **Cara**  **Y-**  **X+** | **Z+**  **Cara X-**  **Y+** | **Cara**  **Y- Z+** | **X-**  **Cara**  **Y+**  **Z-** |
| **Cara**  **Z-** | **Cara X+** |

|  |
| --- |
| Caras ocultas: Caras visibles:  /| N N N N N | / B B B B B/M|  /R| N N N N N | / B B B B B/MM|  /RR| N N N N N | / B B B B B/MMM|  /RRR| N N N N N | / B B B B B/MMMM|  /RRRR| N N N N N | / B B B B B/MMMMM|  |RRRRR/ V V V V V/ |A A A A A |MMMM/  |RRRR/ V V V V V/ |A A A A A |MMM/  |RRR/ V V V V V/ |A A A A A |MM/  |RR/ V V V V V/ |A A A A A |M/  |R/ V V V V V/ |A A A A A |/ |
| 01234567890123456789012345678901234567890123456789 |



A continuación, se realizarán ***20*** giros aleatorios para configurar el cubo en un estado desordenado preparado para su resolución, presentándose el mismo en pantalla. Finalmente, para resolver el cubo se repetirán de forma indefinida los dos siguientes pasos hasta que el cubo se encuentre en un estado ordenado:

* Seleccionar un movimiento de giro, mediante la introducción por teclado del eje de giro (entero en intervalo ***[0,2]***), la capa ó nivel a girar (entero en intervalo ***[0,n-1]***) y el ángulo de giro (entero en intervalo ***[1,3]*** que representa el nº de giros sucesivos de ***90º*** a realizar).
* Imprimir en pantalla el estado resultante del movimiento seleccionado.

|  |  |
| --- | --- |
| Funciones para generar números aleatorios en C | |
| #include <stdlib.h>  #include <time.h> |  |
| time\_t t;  srand((unsigned) time(&t)); | Inicializa el generador de números aleatorios con un valor aleatorio obtenido del reloj del sistema |
| rand()%(b-a+1)+a | Genera un número aleatorio entero en intervalo [a,b]=a,a+1,a+2,...,b-1,b |
| a+(b-a)\*1.0\*rand()/RAND\_MAX | Genera un número aleatorio real en intervalo [a,b] |

|  |  |
| --- | --- |
| **Estrategia de resolución del problema** | |
| Análisis  (Pre-diseño) | Leer enunciado detenidamente e identificar (mediante análisis gramatical de la narrativa del enunciado):   * Información a procesar (sustantivos) 🡪 separar los elementos de E y de S, asignarles un identificador, un tipo de datos y posibles restricciones en sus valores. * Tareas de manipulación de información (verbos, frases verbales) 🡪 Lista de tareas 🡪 descomponer las tareas complejas en tareas más simples ampliando sus descripciones incorporando detalles. |
| Diseño 🡪 2 pasos | 1) Diseño preliminar o arquitectónico   * Diseño de datos: nuevas tipologías * Estructura del programa🡪 diagrama de módulos   1 módulo por cada tarea diferente  Establece relaciones de dependencia (no de control)   * Interfaces entre módulos: para cada módulo del diagrama:   Nombre o identificador  Breve descripción de su actividad funcional  Lista de parámetros formales  2) Diseño detallado o procedimental:  módulo 🡪 sub-algoritmo |

**ANÁLISIS/PRE-DISEÑO**

Información:

E: longitud lado del cubo: n (entero: [1,10]

Movimiento de giro: eje de giro (entero: [0,2] 🡪 0: X+, 1: Y+, 2: Z+)

capa o nivel (entero: [0,n-1])

ángulo de giro (entero: [1,3] 🡪 1: 90º, 2: 180º, 3: 270º)

S: cubo de Rubik: cr (6 caras, cada cara compuesta de nxn colores representados por caracteres

y restringidos al conjunto de colores: Azul, Amarillo, Blanco, Negro, Rojo, Verde)🡪

6 Tablas de caracteres organizados por filas y columnas

Mensaje de terminación: "cubo de Rubik resuelto"

Tareas de manipulación de la información:

* Leer longitud del lado del cubo **n**.
* Inicializar cubo a su configuración de colores inicial.
* Imprimir caras del cubo en pantalla.
* Desordenar el cubo con 20 giros aleatorios. Para cada giro:
  + Generar parámetros de giro de forma aleatoria (eje, capa, ángulo).
  + Girar capa correspondiente del cubo.
    - Girar cara (si capa es **0** ó **n-1**)
* Resolver cubo de Rubik. Repetir las siguientes operaciones hasta que el cubo esté ordenado:
  + Leer por teclado parámetros de giro: eje, capa, ángulo.
  + Girar capa correspondiente del cubo.
    - Girar cara (si capa es **0** ó **n-1**)
  + Imprimir caras del cubo en pantalla.
  + Comprobar que el cubo está resuelto.

**DISEÑO PRELIMINAR**

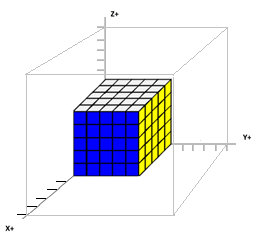
**DISEÑO DE DATOS**

Objetos complejos de información identificados en el análisis del problema:

* Movimiento de giro: (3 enteros)
  + Eje de giro (entero: [0,2] 🡪 0: X+, 1: Y+, 2: Z+)
  + Capa o nivel (entero: [0,n-1])
  + Ángulo de giro (entero: [1,3] 🡪 1: 90º, 2: 180º, 3: 270º)

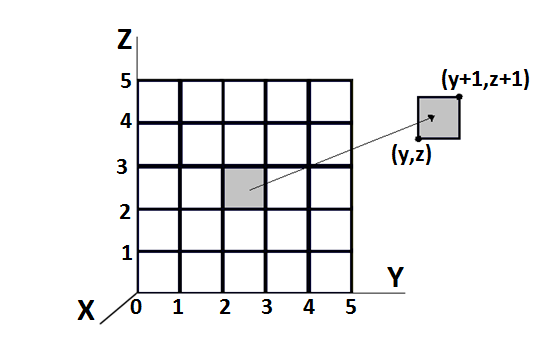
Dos alternativas de diseño: "array" de 3 enteros ó registro con tres campos enteros 🡪la segunda alternativa puede resultar más conveniente ya que los tres datos enteros representan elementos de información muy diferentes, es más sencillo identificarlos por su nombre y no por un índice, y a los que difícilmente se les realizará la misma operación.

* Cubo de Rubik🡪aunque teóricamente está formado por **n3** piezas, para este problema solo nos interesa la información de los colores de las caras visibles de las piezas situadas en sus caras externas (representación de fronteras) 🡪 tenemos 6 caras, cada cara compuesta de nxn colores restringidos al conjunto de colores: Azul, Amarillo, Blanco, Negro, Rojo, Verde. Por simplicidad, representaremos los colores mediante caracteres (‘A’: azul, ‘M’: amarillo, ‘B’: blanco, ‘N’: negro, ‘R’: rojo, ‘V’: verde). Alternativas de diseño:
  + "Array" 3D de 10x10x10 de piezas + entero que indique la longitud (lado) del cubo 🡪 el cubo de lado **n** estaría representado por un “subarray” 3D de piezas que ocuparían las primeras posiciones de los 3 índices 🡪 aunque esta alternativa puede parecer la más realista, habría que tener en cuenta que cada pieza (cubo unitario) tiene 6 caras que habría que representar y procesar, aunque muchos de esos elementos de información no fueran visibles.



La identificación de cada pieza del cubo se realizaría a partir de las coordenadas de intersección de las líneas de cuadrícula de su esquina inferior izquierda, de acuerdo con el sistema de referencia mostrado en la figura.

* + Seis tablas (“arrays” bidimensionales) de 10x10 caracteres + entero que indique la longitud (lado) del cubo 🡪 cada cara del cubo de lado **n** estaría representada por un “subarray” bidimensional de caracteres que ocuparía las **n** primeras filas y columnas.



La identificación de cada pieza en esta cara se realizaría a partir de las coordenadas de intersección de la línea de cuadrícula de su esquina inferior izquierda, que se harían corresponder con el nº de “fila” (primer índice) y nº de “columna” (segundo índice) del “array” bidimensional, dependiendo de la cara considerada. Esto habría que dejarlo perfectamente especificado para poder identificar sin ninguna ambigüedad cada pieza. Para poder explotar la simetría, adoptamos el criterio de identificar las caras (filas/columnas) tomando como punto de referencia el vector normal a las mismas. Partiendo de la cara X+, se obtienen las otras caras mediante el intercambio de los ejes en orden: X🡪Y🡪Z🡪X (0🡪1🡪2🡪0):

|  |  |  |
| --- | --- | --- |
| Cara X+: (y,z)  Z+  Y+ | Cara Y+: (z,x)  X+  Z+ | Cara Z+: (x,y)  Y+  X+ |
| Cara X-: (y,z)  Z+  Y+ | Cara Y-: (z,x)  X+  Z+ | Cara Z-: (x,y)  Y+  X+ |

Observamos que en todos los casos, el primer índice es la abscisa (que corresponde con el nº de columna) y el segundo índice es la ordenada (que se corresponde con el nº de fila). Las “filas” siempre van de abajo hacia arriba, pero las “columnas” en los tres primeros casos van de izquierda a derecha y en los otros tres de derecha a izquierda.

* + Vector ("Array" unidimensional) de 6 tablas de caracteres + entero que indique la longitud (lado) del cubo 🡪esta alternativa, que es muy similar a la anterior, nos permitiría poder explotar la simetría jugando con el significado de los índices (0: X+, 1:Y+, 2: Z+, 3: X-, 4: Y-, 5: Z.), evitando el tener que repetir 3 veces la misma operación de giro (una para cada eje).
  + Registro con dos campos (lado del cubo + vector de 6 tablas) 🡪es similar a la anterior, pero con una única variable representamos el modelo de información (máximo grado de abstracción), lo cual nos va a permitir simplificar las interfaces de las operaciones.🡪 alternativa seleccionada.

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| CONST  MAX=10  MAX\_0=20 | #define MAX 10 // Longitud máxima lado cubo  #define MAX\_0 20 // Nº inicial giros aleatorios |
| TIPOS  tCuboRik: registro de  n: entero  m: tmatriz  Fin\_registro  tmatriz: vector[0..MAX-1,0..MAX-1] de carácter  tGiro: registro de  eje: entero  capa: entero  ang: entero  Fin\_registro | typedef char tmatriz[MAX][MAX];  typedef struct{  int n;  tmatriz m[6];  }tCuboRubik;  typedef struct{  int eje; // 0: X+ 1: Y+ 2: Z+  int capa; // 0..n-1  int ang; // [1,2,3] \* PI/2  }tGiro; |

**ESTRUCTURA DEL PROGRAMA**

Módulo principal

desordenarCubo

resolver Cubo

inicializar Cubo

imprimir Cubo

cubo Resuelto

leer giro

girar Capa Cubo

giro Aleatorio

girar Cara

**INTERFACES ENTRE MÓDULOS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Pseudo-código** | | | | **Sintaxis de C** |
| Nombre módulo | Tipo parám | Nombre parám. | Tipo de datos | /\* Prototipos de funciones \*/ |
| Módulo principal |  |  |  | void inicializarCubo(tCuboRubik \*cr);  void desordenarCubo(tCuboRubik \*cr);  void resolverCubo(tCuboRubik \*cr);  int cuboResuelto(tCuboRubik \*cr);  void leerGiro(int n,tGiro \*g);  void giroAleatorio(int n,tGiro \*g);  void girarCapaCubo(tCuboRubik \*cr,tGiro g);  void girarCara (tmatriz m,int n);  void imprimirCubo(tCuboRubik \*cr); |
| inicializarCubo | S | cr | tCuboRubik |
| desordenarCubo | E/S | cr | tCuboRubik |
| resolverCubo | E/S | cr | tCuboRubik |
| cuboResuelto | E/S | cr | tCuboRubik |
| S |  | lógico |
| leerGiro | E | n | entero |
| S | g | tGiro |
| giroAleatorio | E | n | entero |
| S | g | tGiro |
| girarCapaCubo | E/S | cr | tCuboRubik |
| E | g | tGiro |
| girarCara | E/S | m | tmatriz |
| E | g | tGiro |
| imprimirCubo | E/S | cr | tCuboRubik |

**DISEÑO DETALLADO**

**MÓDULO PRINCIPAL**

* Declarar variable para representar el cubo de Rubik.
* Llamar secuencialmente a los módulos que resuelven los pasos principales del algoritmo: inicializar las caras del cubo a la configuración original de colores, desordenar el cubo (hacer 20 giros aleatorios), imprimir las caras del cubo en pantalla y resolver el cubo.
* Nota: en ***C*** ejecutaremos al principio del programa la función ***srand()*** para revolver la base inicial (semilla) del algoritmo de generación de números aleatorios, inicializándola a un valor obtenido del reloj del sistema.

|  |  |
| --- | --- |
|  |  |
| cuboRubik  crCuboRubik  UBO RUBIK  inicializarCubo(cr)  desordenarCubo(cr)  imprimirCubo(cr)  resolverCubo(cr) | int main(){  char c;  tCuboRubik cr;  time\_t t;  srand((unsigned) time(&t));  do{ system("cls");  printf("CUBO RUBIK\n");  printf("==========\n\n");  inicializarCubo(&cr);  desordenarCubo(&cr);  imprimirCubo(&cr);  resolverCubo(&cr);  printf("\n\Nueva operacion (s/n)? ");  c=toupper(getch());  }while (c!='N');  return 0;  } |







**inicializarCubo**

* Leer por teclado la longitud del cubo (nº de cubos unitarios en que se divide) 🡪 lectura con validación de un entero en intervalo [1-MAX].
* Las caras visibles de los cubos unitarios de cada cara del cubo de Rubik son inicializadas con el color de la configuración por defecto:

|  |  |  |  |
| --- | --- | --- | --- |
| **Cara** | **Vector normal** | **Nomenclatura** | **Color inicial** |
| 0 | X+ | Frontal | A: azul |
| 1 | Y+ | Derecha | M: amarillo |
| 2 | Z+ | Arriba | B: blanco |
| 3 | X- | Trasera | N: negro |
| 4 | Y- | Izquierda | R: rojo |
| 5 | Z- | Abajo | V: verde |

El color es representado por un carácter. Para obtener el color inicial de cada cara, se puede utilizar una selectiva múltiple controlada por el nº de la cara, o alternativamente podríamos utilizar un “array” unidimensional de 6 caracteres inicializado a ‘A’, ‘M’, ‘B’, ‘N’, ‘R’ y ‘V’, ó una cadena de 7 caracteres inicializada a “AMBNRV”.

* Las caras del cubo están representadas por un “array” unidimensional de caras (tablas de caracteres)🡪 recorrido secuencial para colorear uniformemente cada cara🡪 repetición controlada por contador de iteraciones **k**. En cada iteración se asignan los colores correspondientes a cada pieza de la cara:

Desde k=0 Hasta 5 Hacer

{ Colorear piezas cara k }

Fin\_desde

* Para colorear las piezas de cada cara, hay que tener en cuenta que está representada por un “array” bidimensional de los colores (caracteres) de las caras visibles de las diferentes piezas (cubos unitarios)🡪 recorrido secuencial de la tabla de caracteres 🡪 repetición doble controlada por los contadores de iteración **i** y **j** vinculados a los índices que representan las filas y columnas 🡪 en cada iteración del bucle interno se asigna el color de la cara al elemento (i,j) de dicha tabla:

Desde i=0 Hasta cr.n-1 Hacer

Desde j=0 Hasta cr.n-1 Hacer

cr.m[k][i,j]🡨color

Fin\_desde

Fin\_desde

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| Procedimiento inicializarCubo(cr: tCuboRubik (S))  Var color: caracter  i,j,k: entero  Inicio Repetir escribir("Long. cubo (1-",MAX, "):")  leer(cr.n)  Hasta\_Que (cr.n≥1)y(cr.n≤MAX)  Desde k=0 Hasta 5 Hacer  Según\_sea (k) Hacer  0: color🡨'A'  1: color🡨'M'  2: color🡨'B'  3: color🡨'N'  4: color🡨'R'  5: color🡨'V'  Fin\_según\_sea  Desde i=0 Hasta cr.n-1 Hacer  Desde j=0 Hasta cr.n-1 Hacer  cr.m[k][i,j]🡨color  Fin\_desde  Fin\_desde  Fin\_desde  Fin\_Procedimiento | void inicializarCubo(tCuboRubik \*cr){  int i,j,k;  char color[7]="AMBNRV";  do{ printf("Longitud del cubo (1-%d): ",MAX);  scanf(" %d",&cr->n);  }while((cr->n<=0)||(cr->n>MAX));  for(k=0;k<6;++k)  for(i=0;i<cr->n;++i)  for(j=0;j<cr->n;++j)  cr->m[k][i][j]=color[k];  } |

**desordenarCubo**

* Realizar ***20*** giros aleatorios para configurar el cubo en un estado desordenado preparado para su resolución🡪 repetición controlada por contador de iteraciones. En cada iteración:
* Generar de forma aleatoria los parámetros de un giro (eje, capa, ángulo).
* Efectuar el giro de la capa indicada

|  |  |
| --- | --- |
|  |  |
| Procedimiento desordenarCubocr: tCuboRubik (E/S))  ientero  g tGiro  1Max\_0  giroAleatorio(cr.n,g)  girarCapaCubo(cr,g)  procedimiento | void desordenarCubo(tCuboRubik \*cr){  int i;  tGiro g;    for(i=0;i<MAX\_0;++i){  giroAleatorio(cr->n,&g);  girarCapaCubo(cr,g);  }  } |

**resolverCubo**

* Realizar giros sucesivos cuyos parámetros son introducidos por teclado hasta tener resuelto el cubo🡪 repetición

|  |  |
| --- | --- |
| ¿Está el cubo resuelto? | F: leer por teclado parámetros de giro  Girar capa del cubo  Imprimir en pantalla cubo |
| ¿Está el cubo resuelto? | F: leer por teclado parámetros de giro  Girar capa del cubo  Imprimir en pantalla cubo |
| ¿Está el cubo resuelto? | F: leer por teclado parámetros de giro  Girar capa del cubo  Imprimir en pantalla cubo |
| …. | … |
| ¿Está el cubo resuelto? | V |

* + Cuerpo del bucle: leer giro, girar capa e imprimir cubo
  + cubo esté resuelto🡪 el invariante (condición del mientras) es lo contrario de la condición de salida

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| Procedimiento resolverCubo(cr: tCuboRubik (E/S))  Var g: tGiro  Inicio Mientras(no cuboResuelto(cr)) Hacer  leerGiro(cr.n,g)  girarCapaCubo(cr,g)  imprimirCubo(cr)  Fin\_mientras  Fin\_procedimiento | void resolverCubo(tCuboRubik \*cr){  tGiro g;  while(!cuboResuelto(cr)){  leerGiro(cr->n,&g);  girarCapaCubo(cr,g);  imprimirCubo(cr);  }  } |

**giroAleatorio**

* Obtener mediante el generador de números aleatorios los valores de los parámetros de un giro aleatorio:
  + Eje de giro: entero en intervalo [0,2] (0: X+, 1: Y+, 2: Z+).
  + Capa o nivel: entero en intervalo [0,n-1].
  + Ángulo de giro: entero en intervalo [1,3] (1: PI/2, 2: PI, 3: 3\*PI/2).

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| Procedimiento giroAleatorio(n: entero (E),g: tGiro (S))  Inicio g.eje🡨rand() MOD 3  g.capa🡨rand() MOD n  g.ang🡨rand() MOD 3+1  Fin\_procedimiento | void giroAleatorio(int n,tGiro \*g){  g->eje=rand()%3;  g->capa=rand()%n;  g->ang=rand()%3+1;  } |

**leerGiro**

* Leer por teclado con validación los valores de los parámetros del siguiente giro a efectuar para resolver el cubo:
  + Leer eje de giro: entero en intervalo [0,2] (0: X+, 1: Y+, 2: Z+).
  + Leer capa o nivel: entero en intervalo [0,n-1].
  + Leer ángulo de giro: entero en intervalo [1,3] (1: PI/2, 2: PI, 3: 3\*PI/2).
* Nota: para facilitar al usuario la identificación del eje de giro, se va a leer un carácter que luego se convierte en el nº de eje correspondiente mediante una selectiva múltiple.

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| Procedimiento leerGiro(n: entero (E), g: tGiro (S))  Var c: carácter  Inicio Repetir Escribir("Eje de Giro (X,Y,Z): ")  Leer(c)  Según\_sea (c) Hacer  'x', 'X': g.eje🡨0  'y', 'Y': g.eje🡨1  'z', 'Z': g.eje🡨2  Sino g.eje🡨-1  Fin\_según\_sea  Hasta\_que ((g.eje≥0)y(g.eje≤2))  Repetir Escribir("Capa a girar (0-",n-1, "):")  Leer(g.capa)  Hasta\_que((g.capa≥0)y(g.capa≤n-1)  Repetir Escribir("Ang. giro (1:PI/2, 2:PI,3:3\*PI/2): ")  Leer(g.ang)  Hasta\_que ((g.ang≥1)y(g.ang≤3))  Fin\_procedimiento | void leerGiro(int n,tGiro \*g){  char c;  do{ printf("Eje de Giro (X,Y,Z): ");  scanf(" %c",&c);  switch(c){  case 'x':  case 'X': g->eje=0;  break;  case 'y':  case 'Y': g->eje=1;  break;  case 'z':  case 'Z': g->eje=2;  break;  default: g->eje=-1;  }  }while((g->eje<0)||(g->eje>2));  do{ printf("Capa a girar (0-%d): ",n-1);  scanf(" %d",&g->capa);  }while((g->capa<0)||(g->capa>n-1));  do{ printf("Ang. giro (1:PI/2, 2:PI,3:3\*PI/2): ");  scanf(" %d",&g->ang);  }while((g->ang<1)||(g->ang>3));  } |

**cuboResuelto**

* Comprobar que todas las piezas de cada una de las 6 caras son del mismo color🡪 como las 6 caras están en un vector de caras, habrá que hacer un recorrido secuencial del vector: repetición controlada por contador de iteraciones a la que le añadimos una segunda condición de salida controlada por un dato lógico que se activa cuando hay una cara en la que no todas las piezas son de mismo color:

res🡨verdadero

k🡨0

Mientras((k<6)y (res)) Hacer

{ comprobar colores cara k }

Si(res)

Entonces k🡨k+1

Fin\_si

Fin\_mientras







Para comprobar que todas las piezas de una cara tienen el mismo color, bastaría comparar los colores de todas las piezas con el color de la primera 🡪como cada cara es un “array” bidimensional de caracteres, habrá que hacer un recorrido secuencial del mismo 🡪repetición doble controlada por los contadores de iteración **i** y **j** vinculados a los índices que representan las filas y columnas, a los que se le añade una segunda condición de salida controlada por el mismo dato lógico **res** que se activa al encontrar una pieza con un color diferente de la primera🡪 en cada iteración del bucle interno se compara el color de la pieza (i,j) con el de la primera pieza (0,0) en la cara correspondiente:

i🡨0

Mientras((i<cr.n)y(res))

Hacer j🡨0

Mientras((j<cr.n)y(res))

Hacer Si(cr.m[k][i][j]≠ cr.m[k][0][0])

Entonces res🡨falso

Sino j🡨j+1

Fin\_si

Fin\_mientras

Si(res)

Entonces i🡨i+1

Fin\_si

Fin\_mientras

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| Función cuboResuelto(cr: tCuboRubik (E)):lógico  Var i,j,k: entero  res: lógico  Inicio res🡨verdadero  k🡨0  Mientras((k<6)y (res)) Hacer  i🡨0  Mientras((i<cr.n)y(res))  Hacer j🡨0  Mientras((j<cr.n)y(res))  Hacer Si(cr.m[k][i][j]≠cr.m[k][0][0])  Entonces res🡨falso  Sino j🡨j+1  Fin\_si  Fin\_mientras  Si(res)  Entonces i🡨i+1  Fin\_si  Fin\_mientras  Si(res)  Entonces k🡨k+1  Fin\_si  Fin\_mientras  Devolver res  Fin\_procedimiento | int cuboResuelto(tCuboRubik \*cr){  int res,i,j,k;  res=1;  k=0;  while((k<6)&&res){  i=0;  while((i<cr->n)&&res){  j=0;  while((j<cr->n)&&res){  if(cr->m[k][i][j]!=cr->m[k][0][0])  res=0;  else ++j;  }  if(res)  ++i;  }  if(res)  ++k;  }  return res;  } |

**girarCapaCubo**

|  |  |  |  |
| --- | --- | --- | --- |
| **Estado inicial** | **Capa 1 eje X+** | **Giro 90º capa 1** | **Estado final** |
|  |  |  |  |

En el ejemplo se muestra un giro de 90º respecto al eje X+ de la capa nº 1. Este giro involucra el intercambio en orden cíclico anti-horario de cuatro vectores correspondientes a cuatro caras paralelas al eje X. Para giros alrededor de otro eje, se aplica simetría mediante el intercambio de los ejes en orden: X🡪Y🡪Z🡪X (0🡪1🡪2🡪0). En el siguiente cuadro se muestran las tres posibilidades según eje de giro, indicándose en cada caso los vectores involucrados:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Eje | Giro 90º anti-horario nivel n | | | |
| X+ | Cara Z-: columna n | Cara Y+: fila n | Cara Z+: columna n | Cara Y-: fila n |
| Y+ | Cara X-: columna n | Cara Z+: fila n | Cara X+: columna n | Cara Z-: fila n |
| Z+ | Cara Y-: columna n | Cara X+: fila n | Cara Y+: columna n | Cara X-: fila n |

Solo habría que resolver uno de los tres casos (por ejemplo el primero). Si numeramos los ejes, nos resultará más sencillo identificarlos y aplicar la simetría. Por ejemplo, para obtener los índices de los tres ejes positivos, se podría utilizar:

ejex🡨g.eje

ejey🡨(g.eje+1) MOD 3

ejez🡨(g.eje+2) MOD 3

Para los índices de los ejes negativos, bastaría con sumar 3 al índice del correspondiente eje positivo.

Para copiar los vectores se utiliza una repetición controlada por contador de iteraciones, dónde el índice i se asocia a las respectivas posiciones del vector columna/fila de la correspondiente cara. Hay que tener en cuenta:

* Columna/fila se refieren respectivamente al primer/segundo índice del "array" bidimensional que representa la cara.
* Las columnas se numeran de abajo hacia arriba, pero las filas se numeran de izquierda a derecha las de las caras correspondientes a los ejes positivos y de derecha a izquierda las de los ejes negativos.
* Se requiere una variable auxiliar de tipo carácter para hacer los intercambios.

Desde i=0 Hasta cr.n-1 Hacer

aux🡨cr.m[ejez+3][g.capa][i]

cr.m[ejez+3][g.capa][i]🡨cr.m[ejey+3][cr.n-1-i][g.capa]

cr.m[ejey+3][cr.n-1-i][g.capa]🡨cr.m[ejez][g.capa][cr.n-1-i]

cr.m[ejez][g.capa][cr.n-1-i]🡨cr.m[ejey][i][g.capa]

cr.m[ejey][i][g.capa]🡨aux

Fin\_desde

Si el ángulo de rotación es de 180º ó 270º, tan solo habría que repetir 2 ó 3 veces el giro de 90º:

Desde j=1 Hasta g.ang Hacer

{ Realizar giro de 90º }

Fin\_desde

Finalmente, habría que considerar la posibilidad de que la capa a girar fuera la de un extremo, en cuyo caso habrá que girar la cara correspondiente:

Si(g.capa=0)

Entonces Desde i=1 Hasta g.ang Hacer

girarCara(cr.m[ejex+3],cr.n)

Fin\_desde

Sino Si(g.capa=cr.n-1)

Entonces Desde i=1 Hasta g.ang Hacer

girarCara(cr.m[ejex],cr.n)

Fin\_desde

Fin\_si

Fin\_si

|  |
| --- |
| **Pseudo-código** |
| Procedimiento girarCapaCubo(cr: tCuboRubik (E/S), g: tGiro (E))  Var i,j: entero  aux: carácter  ejex,ejey,ejez: entero  Inicio ejex🡨g.eje  ejey🡨(g.eje+1) MOD 3  ejez🡨(g.eje+2) MOD 3  Si(g.capa=0)  Entonces Desde i=1 Hasta g.ang Hacer  girarCara(cr.m[ejex+3],cr.n)  Fin\_desde  Sino Si(g.capa=cr.n-1)  Entonces Desde i=1 Hasta g.ang Hacer  girarCara(cr.m[ejex],cr.n)  Fin\_desde  Fin\_si  Fin\_si  Desde j=1 Hasta g.ang Hacer  Desde i=0 Hasta cr.n-1 Hacer  aux🡨cr.m[ejez+3][g.capa][i]  cr.m[ejez+3][g.capa][i]🡨cr.m[ejey+3][cr.n-1-i][g.capa]  cr.m[ejey+3][cr.n-1-i][g.capa]🡨cr.m[ejez][g.capa][cr.n-1-i]  cr.m[ejez][g.capa][cr.n-1-i]🡨cr.m[ejey][i][g.capa]  cr.m[ejey][i][g.capa]🡨aux  Fin\_desde  Fin\_desde  Fin\_procedimiento |
| **Sintaxis de C** |
| void girarCapaCubo(tCuboRubik \*cr,tGiro g){  int i,j;  char aux;  int ejex,ejey,ejez;    ejex=g.eje;  ejey=(g.eje+1)%3;  ejez=(g.eje+2)%3;  if(g.capa==0)  for(i=0;i<g.ang;++i)  girarCara(cr->m[ejex+3],cr->n);  else if(g.capa==cr->n-1)  for(i=0;i<g.ang;++i)  girarCara(cr->m[ejex],cr->n);  for(j=0;j<g.ang;++j)  for(i=0;i<cr->n;++i){  aux=cr->m[ejez+3][g.capa][i];  cr->m[ejez+3][g.capa][i]=cr->m[ejey+3][cr->n-1-i][g.capa];  cr->m[ejey+3][cr->n-1-i][g.capa]=cr->m[ejez][g.capa][cr->n-1-i];  cr->m[ejez][g.capa][cr->n-1-i]=cr->m[ejey][i][g.capa];  cr->m[ejey][i][g.capa]=aux;  }  } |

**girarCara**

* Recorrido secuencial del “array” bidimensional para calcular cada elemento del “array” girado 🡪 repetición doble controlada por contadores de iteraciones i (columnas) y j (filas)
* Planteemos un ejemplo para analizar las operaciones a realizar cuando giramos un “array” bidimensional 90º en sentido anti-horario con pivote en la esquina inferior izquierda:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 5 | a | b | c | d | e | f |
| 4 | g | h | i | j | k | l |
| 3 | m | n | o | p | q | r |
| 2 | s | t | u | v | w | x |
| 1 | y | z | 0 | 1 | 2 | 3 |
| 0 | 4 | 5 | 6 | 7 | 8 | 9 |
|  | 0 | 1 | 2 | 3 | 4 | 5 |

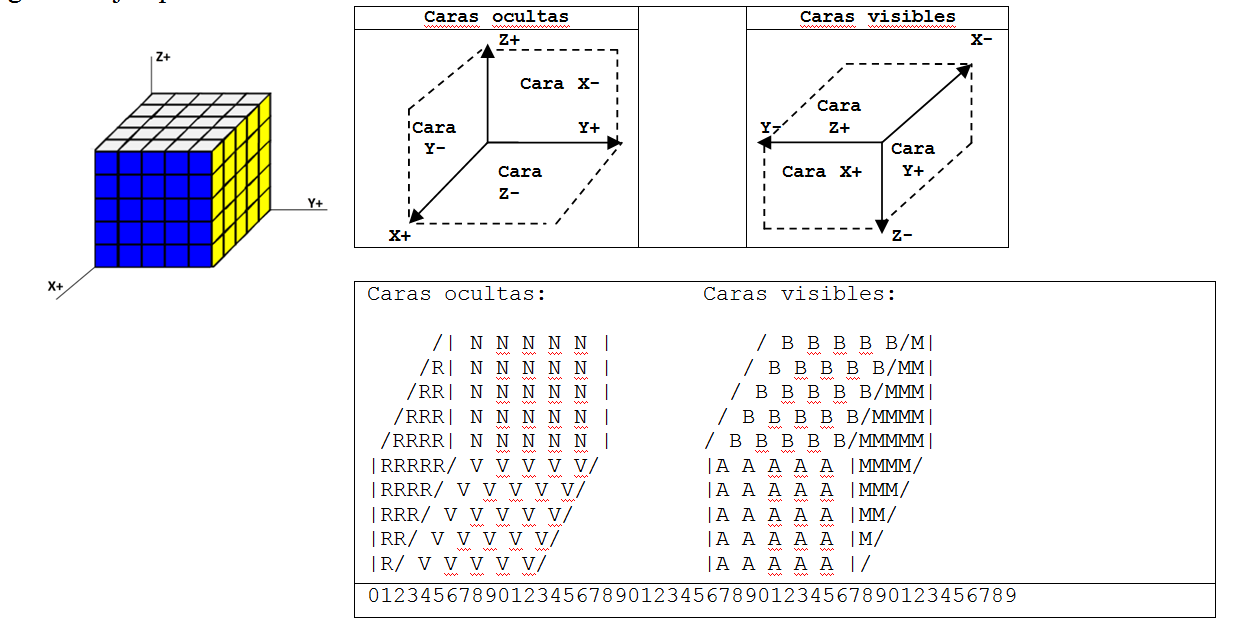
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 5 | f | l | r | x | 3 | 9 |
| 4 | e | k | q | w | 2 | 8 |
| 3 | d | j | p | v | 1 | 7 |
| 2 | c | i | o | u | 0 | 6 |
| 1 | b | h | n | t | z | 5 |
| 0 | a | g | m | s | y | 4 |
|  | 0 | 1 | 2 | 3 | 4 | 5 |

* El nº de columna pasa a ser directamente el nº de fila, y el nº de fila pasa a ser el de columna pero empezando a contar por la columna de la derecha: (i,j) 🡪 (n-1-j,i).
* Utilizar un “array” auxiliar para realizar la operación de giro (ya que al sobre-escribirla se perderían algunos valores originales necesarios para calcular otros valores del “array” girado), y al finalizarla se copia al “array” original.
* Nota: hay que tener en cuenta que el sistema de coordenadas cartesianas discreto que estamos utilizando, la columna es el primer índice y la fila es el segundo índice.

|  |  |
| --- | --- |
| **Pseudo-código** | **Sintaxis de C** |
| Procedimiento girarCara(m: tmatriz m (E/S), n: entero (E))  Var aux: tmatriz  i,j: entero  Inicio Desde i=0 Hasta n-1 Hacer  Desde j=0 Hasta n-1 Hacer  aux[i][j]🡨m[j][n-1-i]  Fin\_desde  Fin\_desde  Desde i=0 Hasta n-1 Hacer  Desde j=0 Hasta n-1 Hacer  m[i][j]🡨aux[i][j]  Fin\_desde  Fin\_desde  Fin\_procedimiento | void girarCara(tmatriz m,int n){  tmatriz aux;  int i,j;  for(i=0;i<n;++i)  for(j=0;j<n;++j)  aux[i][j]=m[j][n-1-i];  for(i=0;i<n;++i)  for(j=0;j<n;++j)  m[i][j]=aux[i][j];  } |

**imprimirCubo**

Vamos a plantear inicialmente la impresión de las 6 caras del cubo de forma separada. Cada cara del cubo está representada por un "array" bidimensional de caracteres 🡪 recorrido secuencial doble controlado por dos contadores de iteraciones i y j que se vinculan con los índices correspondientes del "array". Tan solo hay que tener en cuenta el significado de los índices en función de la cara y del punto de vista.



Empezaremos por las caras X+ y X- que son las más sencillas de imprimir, ya que se presentan en forma cuadrada. Hay que tener en cuenta el segundo índice **i** representa la fila (de abajo hacia arriba) y el primer índice **j** representa la columna (izquierda a derecha):

|  |  |
| --- | --- |
| Cara X- | Cara X+ |
| Desde i=0 Hasta cr.n-1 Hacer  Escribir("|")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr.m[3][j][cr.n-1-i])  Fin\_desde  Escribir(" |")  Escribir("\n")  Fin\_mientras | Desde i=0 Hasta cr.n-1 Hacer  Escribir(" |")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr.m[0][j][cr.n-1-i])  Fin\_desde  Escribir("|")  Escribir("\n")  Fin\_procedimiento |

Continuaremos por las caras Z+ y Z- que son similares a las anteriores con la excepción de los espacios en blanco que hay que situar al final/principio de cada línea respectivamente y que fácilmente se relacionan con el nº de fila (en un caso coincide y en el otro va al revés). También hay que tener en cuenta el punto de vista desde el que se está imprimiendo la imagen: el primer índice (columna ó coordenada x) del "array" pasa a ser la fila al imprimir:

|  |  |
| --- | --- |
| Cara Z- | Cara Z+ |
| Desde i=0 Hasta cr.n-1 Hacer  { Cara Z- }  Escribir("/")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr.m[5][i][j])  Fin\_desde  Escribir("/")  { Espacios en blanco }  Desde j=0 Hasta i Hacer  Escribir(" ")  Fin\_desde  Fin\_mientras | Desde i=0 Hasta cr.n-1 Hacer  { Espacios en blanco }  Desde j=0 Hasta cr.n-i Hacer  Escribir(" ")  Fin\_desde  { Cara Z+ }  Escribir("/")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr->m[2][i][j])  Fin\_desde  Escribir("/")  Fin\_procedimiento |

Terminaremos por las caras Y+ e Y- que son las más complicadas de imprimir ya que se imprimen en diagonal (forma de rombo). En cada línea se imprime una diagonal y el nº de diagonales es 2\*n-1 siendo n la longitud del lado del cubo. El rombo lo vamos a imprimir en dos partes, a diagonal por línea, vinculando en cada parte el nº de línea con la diagonal a imprimir:

Desde i=0 Hasta cr.n-1 Hacer

{ Diagonal i Cara Y- (parte superior) }

{ Diagonal i Cara Y+ (parte superior) }

Fin\_mientras

Desde i=0 Hasta cr.n-1 Hacer

{ Diagonal i Cara Y- (parte inferior) }

{ Diagonal i Cara Y+ (parte inferior) }

Fin\_mientras

Como el nº de diagonales es impar, una parte tendrá una diagonal más que la otra, dependiendo de la cara a la que pertenezca.

Cada diagonal consta de un nº de elementos del "array" y se imprime con una repetición simple controlada por un contador de iteraciones. Para movernos en el "array" bidimensional en diagonal hay que incrementar/decrementar ambos índices de forma simultánea en cada iteración. Tan solo necesitamos conocer el nº de elementos de la diagonal y en qué posición del "array" empieza. Por ejemplo, para imprimir una diagonal de la parte superior de las caras Y- e Y+:

{ Cara Y- (parte superior) }

**X+**

**3**

**2**

**1**

**0**

c🡨c0-1

f🡨f0

Mientras(c>=0) Hacer

Escribir(cr.m[4][f][c])

f🡨f-1

c🡨c-1

Fin\_mientras

**3 2 1 0 Z+**

{ Cara Y+ (parte superior) }

c🡨c0

f🡨f0

Mientras(c>=0)

Escribir(cr->m[1][f][c])

f🡨f-1

c🡨c-1

Fin\_mientras

Nótese que el nº de columna coincide con el nº de elementos de la diagonal y que la cara negativa tiene una diagonal menos que la cara positiva (para la parte inferior pasa al contrario)

Para la siguiente diagonal, tan solo hay que incrementar c0 en una unidad. Para la primera diagonal, habría que hacer la siguiente inicialización:

f0🡨cr.n-1

c0🡨0

Para imprimir las diagonales de las partes inferiores, el proceso sería similar, cambiando las inicializaciones de las diagonales y el control de la diagonal que ahora se realiza con la fila

|  |  |
| --- | --- |
| Parte superior caras Y- e Y+ | Parte inferior caras Y- e Y+ |
| f0🡨cr.n-1  c0🡨0  Desde i=0 Hasta cr.n-1 Hacer  { Espacios en blanco }  Desde j=0 Hasta cr.n-i Hacer  Escribir(" ")  Fin\_desde  { Cara Y- (parte superior) }  Escribir("/")  c🡨c0-1  f🡨f0  Mientras(c>=0) Hacer  Escribir(cr.m[4][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  { Cara Y+ (parte superior) }  c🡨c0  f🡨f0  Mientras(c>=0)  Escribir(cr->m[1][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  c0🡨c0+1  Escribir("|")  Escribir("\n")  Fin\_mientras | c0🡨cr.n-1  f0🡨cr.n-2  Desde i=0 Hasta cr.n-1 Hacer  { Cara Y- (parte inferior) }  Escribir(" |")  c🡨c0  f🡨f0+1  Mientras(f>=0) Hacer  Escribir(cr.m[4][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  { Cara Y+ (parte inferior) }  c🡨c0  f🡨f0  Mientras(f>=0) Hacer  Escribir(cr.m[1][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  f0🡨f0-1  Escribir("/")  Escribir("\n")  Fin\_procedimiento |

|  |
| --- |
| **Pseudo-código** |
| Procedimiento imprimirCubo(cr: tCuboRubik (E/S))  Var i,j,f,c,f0,c0: entero  Inicio Escribir(" Caras ocultas Caras visibles:")  { Caras visibles }  f0🡨cr.n-1  c0🡨0  Desde i=0 Hasta cr.n-1 Hacer  { Espacios en blanco }  Desde j=0 Hasta cr.n-i Hacer  Escribir(" ")  Fin\_desde  { Cara Y- (parte superior) }  Escribir("/")  c🡨c0-1  f🡨f0  Mientras(c>=0) Hacer  Escribir(cr.m[4][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  { Cara x- }  Escribir("|")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr->m[3][j][cr->n-1-i])  Fin\_desde  Escribir(" |")  { Separacion entre vistas }  Escribir(" ")  { Espacios en blanco }  Desde j=0 Hasta cr.n-i Hacer  Escribir(" ")  Fin\_desde  { Cara Z+ }  Escribir("/")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr->m[2][i][j])  Fin\_desde  Escribir("/")  { Cara Y+ (parte superior) }  c🡨c0  f🡨f0  Mientras(c>=0)  Escribir(cr->m[1][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  c0🡨c0+1  Escribir("|")  Escribir("\n")  Fin\_mientras  c0🡨cr.n-1  f0🡨cr.n-2  Desde i=0 Hasta cr.n-1 Hacer  { Cara Y- (parte inferior) }  Escribir(" |")  c🡨c0  f🡨f0+1  Mientras(f>=0) Hacer  Escribir(cr.m[4][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  { Cara Z- }  Escribir("/")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr.m[5][i][j])  Fin\_desde  Escribir("/")  { Espacios en blanco }  Desde j=0 Hasta i Hacer  Escribir(" ")  Fin\_desde  { Separacion entre vistas }  Escribir(" ")  { Cara X+ }  Escribir(" |")  Desde j=0 Hasta cr.n-1 Hacer  Escribir(cr.m[0][j][cr.n-1-i])  Fin\_desde  Escribir("|")  { Cara Y+ (parte inferior) }  c🡨c0  f🡨f0  Mientras(f>=0) Hacer  Escribir(cr.m[1][f][c])  f🡨f-1  c🡨c-1  Fin\_mientras  f0🡨f0-1  Escribir("/")  Escribir("\n")  Fin\_desde  Fin\_procedimiento |

|  |
| --- |
| **Sintaxis de C** |
| void imprimirCubo(tCuboRubik \*cr){  int i,j,f,c,f0,c0;  system("cls");  printf(" Caras ocultas Caras visibles:\n\n\n");  // Caras visibles  f0=cr->n-1;  c0=0;  for(i=0;i<cr->n;i++){  // Espacios en blanco  for(j=0;j<cr->n-i+1;++j)  printf(" ");  // Cara Y- (parte superior)  printf("/");  c=c0-1;  f=f0;  while(c>=0){  printf("%c",cr->m[4][f][c]);  f--;  c--;  }  // Cara x-  printf("|");  for(j=0;j<cr->n;++j)  printf(" %c",cr->m[3][j][cr->n-1-i]);  printf(" |");  // Separacion entre vistas  printf(" ");  // Espacios en blanco  for(j=0;j<cr->n-i+1;++j)  printf(" ");  // Cara Z+  printf("/");  for(j=0;j<cr->n;++j)  printf(" %c",cr->m[2][i][j]);  printf("/");  // Cara Y+ (parte superior)  c=c0;  f=f0;  while(c>=0){  printf("%c",cr->m[1][f][c]);  f--;  c--;  }  c0++;  printf("|");  printf("\n");  }  c0=cr->n-1;  f0=cr->n-2;;  for(i=0;i<cr->n;i++){  // Cara Y- (parte inferior)  printf(" |");  c=c0;  f=f0+1;  while(f>=0){  printf("%c",cr->m[4][f][c]);  f--;  c--;  }  // Cara Z-  printf("/");  for(j=0;j<cr->n;++j)  printf(" %c",cr->m[5][i][j]);  printf("/");  // Espacios en blanco  for(j=0;j<=i;++j)  printf(" ");  // Separacion entre vistas  printf(" ");  // Cara X+  printf(" |");  for(j=0;j<cr->n;++j)  printf("%c ",cr->m[0][j][cr->n-1-i]);  printf("|");  // Cara Y+ (parte inferior)  c=c0;  f=f0;  while(f>=0){  printf("%c",cr->m[1][f][c]);  f--;  c--;  }  f0--;  printf("/");  printf("\n");  }  } |

**CRITERIOS UTILIZADOS EN LA EVALUACIÓN DEL EXAMEN**

A modo orientativo, para la evaluación de este examen de convocatoria se desglosó la calificación en base a los sub-problemas principales a resolver descritos en la siguiente tabla. Cualquier alternativa de solución creativa e innovadora para algún aspecto del problema planteada por el alumno, fue tenida en cuenta positivamente en la evaluación. Aspectos desfavorables en la evaluación: mal dominio de la sintaxis de C (errores sintácticos destacables y continuos), diseño y utilización incorrecta de estructuras de control, de estructuras de datos y de interfaces de módulos.

|  |  |
| --- | --- |
| **Elementos a evaluar** | **Puntos** |
| Diseño preliminar: diseño de datos (nuevas tipologías) + estructura programa (diagrama de módulos) + interfaces entre módulos (prototipos de funciones) | 1 |
| Módulo principal + inicializar cubo a estado ordenado inicial + leer por teclado parámetros de giro | 1 |
| Desordenar cubo + generar giro aleatorio | 1 |
| Imprimir cubo en pantalla | 2 |
| Girar capa del cubo + girar matriz | 2 |
| Comprobar que el cubo está resuelto | 1 |
| Total | 8 |

La calificación final en la asignatura consta de tres apartados cuya contribución es la siguiente:

* Examen final: 80%
* Trabajo en Equipo (Grupo de Trabajo): 10%
* Trabajo Individual: 10%

|  |
| --- |
| **Implementación completa del programa en lenguaje de programación C** |
| #include <stdlib.h>  #include <stdio.h>  #include <conio.h>  #include <ctype.h>  #include <math.h>  #include <time.h>  #define MAX 10 // Longitud máxima del lado del cubo  #define MAX\_0 1 // Nº de giros aleatorios para configuracion inicial del cubo  /\* Diseño Preliminar \*/  /\* Diseño de Datos \*/  /\* Nuevos tipos de datos \*/  typedef char tmatriz[MAX][MAX];  typedef struct{  int n;  tmatriz m[6];  }tCuboRubik;  typedef struct{  int eje; // 0: X+ 1: Y+ 2: Z+  int capa; // 0..n-1  int ang; // [1,2,3] \* PI/2  }tGiro;  /\* Interfaces entre módulos \*/  /\* Prototipos de funciones \*/  void inicializarCubo(tCuboRubik \*cr);  void desordenarCubo(tCuboRubik \*cr);  void resolverCubo(tCuboRubik \*cr);  int cuboResuelto(tCuboRubik \*cr);  void leerGiro(int n,tGiro \*g);  void giroAleatorio(int n,tGiro \*g);  void girarCapaCubo(tCuboRubik \*cr,tGiro g);  void girarCara(tmatriz m,int n);  void imprimirCubo(tCuboRubik \*cr);  /\* Diseño Detallado \*/  /\* Definiciones de funciones \*/  int main(){  char c;  tCuboRubik cr;  time\_t t;  srand((unsigned) time(&t));  do{ system("cls");  printf("CUBO RUBIK\n");  printf("==========\n\n");  inicializarCubo(&cr);  desordenarCubo(&cr);  imprimirCubo(&cr);  resolverCubo(&cr);  printf("\n\nDesea efectuar una nueva operacion (s/n)? ");  c=toupper(getch());  }while (c!='N');  return 0;  }  void inicializarCubo(tCuboRubik \*cr){  int i,j,k;  char color[7]="AMBNRV";  do{ printf("Longitud del cubo (1-%d): ",MAX);  scanf(" %d",&cr->n);  }while((cr->n<=0)||(cr->n>MAX));  for(k=0;k<6;++k)  for(i=0;i<cr->n;++i)  for(j=0;j<cr->n;++j)  cr->m[k][i][j]=color[k];  }  void desordenarCubo(tCuboRubik \*cr){  int i;  tGiro g;    for(i=0;i<MAX\_0;++i){  giroAleatorio(cr->n,&g);  girarCapaCubo(cr,g);  }  }  void resolverCubo(tCuboRubik \*cr){  tGiro g;  while(!cuboResuelto(cr)){  leerGiro(cr->n,&g);  girarCapaCubo(cr,g);  imprimirCubo(cr);  }  }  void giroAleatorio(int n,tGiro \*g){  g->eje=rand()%3;  g->capa=rand()%n;  g->ang=rand()%3+1;  }  void leerGiro(int n,tGiro \*g){  char c;  do{ printf("Eje de Giro (X,Y,Z): ");  scanf(" %c",&c);  switch(c){  case 'x':  case 'X': g->eje=0;  break;  case 'y':  case 'Y': g->eje=1;  break;  case 'z':  case 'Z': g->eje=2;  break;  default: g->eje=-1;  }  }while((g->eje<0)||(g->eje>2));  do{ printf("Capa a girar (0-%d): ",n-1);  scanf(" %d",&g->capa);  }while((g->capa<0)||(g->capa>n-1));  do{ printf("Angulo de giro (1:PI/2, 2:PI,3:3\*PI/2): ");  scanf(" %d",&g->ang);  }while((g->ang<1)||(g->ang>3));  }  int cuboResuelto(tCuboRubik \*cr){  int res,i,j,k;  res=1;  k=0;  while((k<6)&&res){  i=0;  while((i<cr->n)&&res){  j=0;  while((j<cr->n)&&res){  if(cr->m[k][i][j]!=cr->m[k][0][0])  res=0;  else ++j;  }  if(res)  ++i;  }  if(res)  ++k;  }  return res;  }  void girarCapaCubo(tCuboRubik \*cr,tGiro g){  int i,j;  char aux;  int ejex,ejey,ejez;    ejex=g.eje;  ejey=(g.eje+1)%3;  ejez=(g.eje+2)%3;  if(g.capa==0)  for(i=0;i<g.ang;++i)  girarCara(cr->m[ejex+3],cr->n);  else if(g.capa==cr->n-1)  for(i=0;i<g.ang;++i)  girarCara(cr->m[ejex],cr->n);  for(j=0;j<g.ang;++j)  for(i=0;i<cr->n;++i){  aux=cr->m[ejez+3][g.capa][i];  cr->m[ejez+3][g.capa][i]=cr->m[ejey+3][cr->n-1-i][g.capa];  cr->m[ejey+3][cr->n-1-i][g.capa]=cr->m[ejez][g.capa][cr->n-1-i];  cr->m[ejez][g.capa][cr->n-1-i]=cr->m[ejey][i][g.capa];  cr->m[ejey][i][g.capa]=aux;  }  }  void girarCara(tmatriz m,int n){  tmatriz aux;  int i,j;  for(i=0;i<n;++i)  for(j=0;j<n;++j)  aux[i][j]=m[j][n-1-i];  for(i=0;i<n;++i)  for(j=0;j<n;++j)  m[i][j]=aux[i][j];  }  void imprimirCubo(tCuboRubik \*cr){  int i,j,f,c,f0,c0;  system("cls");  printf(" Caras ocultas Caras visibles:\n\n\n");  // Caras visibles  f0=cr->n-1;  c0=0;  for(i=0;i<cr->n;i++){  // Espacios en blanco  for(j=0;j<cr->n-i+1;++j)  printf(" ");  // Cara Y- (parte superior)  printf("/");  c=c0-1;  f=f0;  while(c>=0){  printf("%c",cr->m[4][f][c]);  f--;  c--;  }  // Cara x-  printf("|");  for(j=0;j<cr->n;++j)  printf(" %c",cr->m[3][j][cr->n-1-i]);  printf(" |");  // Separacion entre vistas  printf(" ");  // Espacios en blanco  for(j=0;j<cr->n-i+1;++j)  printf(" ");  // Cara Z+  printf("/");  for(j=0;j<cr->n;++j)  printf(" %c",cr->m[2][i][j]);  printf("/");  // Cara Y+ (parte superior)  c=c0;  f=f0;  while(c>=0){  printf("%c",cr->m[1][f][c]);  f--;  c--;  }  c0++;  printf("|");  printf("\n");  }  c0=cr->n-1;  f0=cr->n-2;  for(i=0;i<cr->n;i++){  // Cara Y- (parte inferior)  printf(" |");  c=c0;  f=f0+1;  while(f>=0){  printf("%c",cr->m[4][f][c]);  f--;  c--;  }  // Cara Z-  printf("/");  for(j=0;j<cr->n;++j)  printf(" %c",cr->m[5][i][j]);  printf("/");  // Espacios en blanco  for(j=0;j<=i;++j)  printf(" ");  // Separacion entre vistas  printf(" ");  // Cara X+  printf(" |");  for(j=0;j<cr->n;++j)  printf("%c ",cr->m[0][j][cr->n-1-i]);  printf("|");  // Cara Y+ (parte inferior)  c=c0;  f=f0;  while(f>=0){  printf("%c",cr->m[1][f][c]);  f--;  c--;  }  f0--;  printf("/");  printf("\n");  }  } |