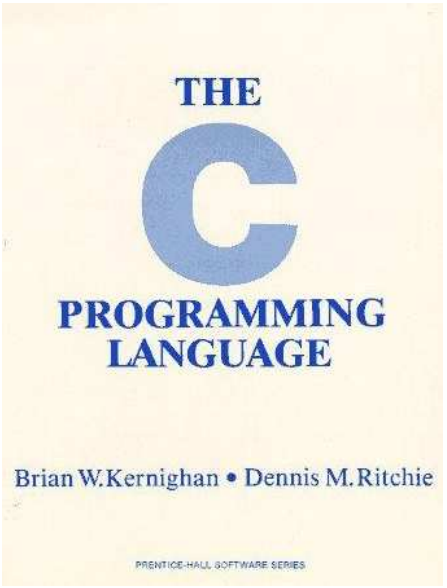
UNIVERSIDAD DE LA CAÑADA

Guía del Concurso de Programación en C

M. C. José Alberto Márquez Domínguez
M. C. Silviana Juárez Chalini
M. C. Beatriz Adriana Sabino Moxo
Dr. Octavio Alberto Agustín Aquino

Carretera Teotitlán - San Antonio Nanahuatipán Km 1.7 s/n. Paraje Titlacuatitla.
Teotitlán de Flores Magón, Oax. México, C.P. 68540

Universidad de la Cañada



El Segundo Concurso de Programación CProg-UNCA, es una competencia que fomenta la creatividad, el trabajo en equipo y la innovación en la creación de programas de cómputo, además de que permite a los estudiantes universitarios probar sus habilidades.

Contenido

INTRODUCCIÓN 3

EN MEMORIA DE DENNIS RITCHIE 4

FASES PARA LA RESOLUCIÓN DE PROBLEMAS 5

Análisis del Problema 6

Diseño del Algoritmo 9

NIVEL 1 BÁSICO: ESTRUCTURAS DE CONTROL SELECTIVAS E ITERATIVAS 10

Ejercicios Resueltos 14

Ejercicios Propuestos 18

NIVEL 2 MEDIO: FUNCIONES Y RECURSIVIDAD 19

Ejercicios Resueltos 22

Ejercicios Propuestos 31

NIVEL 3 AVANZADO: ARREGLOS Y PUNTEROS 33

Ejercicios Resueltos 37

Ejercicios Propuestos 44

INTRODUCCIÓN

C es un lenguaje de nivel medio que actúa con enorme rapidez, tanto en la compilación como en la ejecución de los programas, y que, además, se caracteriza por ser muy portable y fácilmente estructurable. Posee un limitado número de sentencias o palabras clave muy fáciles de memorizar con las que se pueden construir funciones o rutinas que se incorporarán al lenguaje del usuario en forma de nuevas librerías, lo que lo convierte en un lenguaje de muy alto nivel y fácil manejo.

El objetivo de este manual de ejercicios es proveer al estudiante las habilidades y destrezas propias del manejo de un método algorítmico disciplinado, logradas por medio del uso de estructuras abstractas, de control, selectivas, secuenciales y repetitivas, así como utilizar buenas prácticas en las etapas de diseño, codificación, depuración, pruebas de sus aplicaciones.

EN MEMORIA DE DENNIS RITCHIE

(1941-2011)

En 1967 se unió a los laboratorios Bell para empezar una silenciosa revolución informática que por supuesto tiene impacto y consecuencias sumamente positivas hoy día, desde la creación de nuevas plataformas que heredan estilos y formatos de otras, hasta cada vez que un programador declara una variable.

Su pasaje a la historia queda principalmente gobernado por ser el creador del lenguaje C, uno de los puntos de partida del lenguaje de programación moderno y pilar –no sólo simbólico- de la informática actual, aunque Dennis Ritchie también participó protagónicamente del desarrollo de B, Altran, BCPL, Multics y del inefable Unix. Sobre C ya había dicho que era "peculiar, defectuoso y un enorme suceso". Y de UNIX expresó una de las frases más populares de su paso por este mundo: "Unix es simple, sólo hace falta ser un genio para comprender su simplicidad."

Si bien la fama entre los no informáticos y las cuentas bancarias con diez ceros a la derecha no fueron características de su figura, Dennis MacAlistair Ritchie recibió premios como el Turing Award en 1983 por la creación de la teoría de sistemas operativos genéricos, específicamente aplicado a Unix. Otros premios son la envidiada medalla IEEE Richard Hamming, también en función de sus logros con UNIX y la creación de C finalizada en 1973, así como también la medalla Nacional de Tecnología de Estados Unidos puesta en su cuello por el entonces presidente Bill Clinton. Recientemente había recibido el premio de Japón, junto a otro genio, Ken Thompson, principal responsable de B.

FASES PARA LA RESOLUCIÓN DE PROBLEMAS

Las fases o etapas constituyen el ciclo de vida del software, ayudarán en el proceso de resolución de un problema, estas consisten en:

1. Análisis del problema.
2. Diseño del algoritmo.
3. Codificación (Implementación).
4. Compilación y ejecución.
5. Verificación
6. Depuración.
7. Mantenimiento.
8. Documentación.

Las dos primeras etapas conducen a un diseño detallado escrito de forma de algoritmo[1]. Durante la tercera etapa (Codificación) se implementa el algoritmo en un código escrito en un lenguaje de programación reflejando las ideas desarrolladas en las fases de análisis y diseño [Joyanes, 2003].

La Compilación, Ejecución y Verificación realiza la traducción y ejecución del programa, se comprueba rigurosamente y se eliminan todos los errores que pueda tener. Si existen errores es necesario modificarlo y actualízalo de manera que cumplan todas las necesidades de cambio de sus usuarios, para ello se usan las etapas de Verificación y Depuración.

[1] Definido como un conjunto de instrucciones utilizadas para resolver un problema específico.

Finalmente se debe usar la fase de Documentación, es decir, es la escritura de las diferentes fases del ciclo de vida del software, esencialmente el análisis, diseño y codificación, unidos a manuales de usuario y de referencia, así como normas para el mantenimiento.

En este concurso se pondrán a prueba las cuatro primeras fases, aunque se recomienda realizar las fases faltantes para terminar con el ciclo de vida del software.

Análisis del Problema

En esta fase se requiere una clara definición del problema, para poder hacer esto es conveniente realizar las siguientes preguntas:

1. ¿Qué entradas se requieren? (tipo y cantidad)
2. ¿Cuál es la salida deseada? (tipo y cantidad)
3. ¿Qué método produce la salida deseada?

Con dichas preguntas se determina qué necesita el programa para resolver el problema. La solución puede llevarse a cabo mediante varios algoritmos [Joyanes, 2004].

Un algoritmo dado correctamente resuelve un problema definido y determinado.

El algoritmo debe cumplir diferentes propiedades:

1. *Especificación precisa de la entrada.* Se debe dejar claro el número y tipo de valores de entrada y las condiciones iniciales que deben cumplir dichos valores.
2. *Especificación precisa de cada instrucción.* No debe haber ambigüedad sobre las acciones que se deben ejecutar en cada momento.
3. *Exactitud, corrección.* Si debe mostrar que el algoritmo resuelva el problema.

4. *Etapas bien definidas y concretas.* Concreto quiere decir que la acción descrita por esa etapa está totalmente comprendida por la persona o máquina que debe ejecutar el algoritmo. Cada etapa debe ser ejecutable en una cantidad finita de tiempo.
5. *Número finito de pasos.* Un algoritmo se debe componer de un número finito de pasos.
6. *Un algoritmo debe terminar.* En otras palabras, no debe entrar en un ciclo infinito.
7. *Descripción del resultado o efecto.* Debe estar claro cuál es la tarea que el algoritmo debe ejecutar. La mayoría de las veces, esta condición se expresa con la producción de un valor como resultado que tenga ciertas propiedades.

Ejemplo 1

¿Es un algoritmo la siguiente instrucción?

Problema: Escribir una lista de todos los enteros positivos

Solución: Es imposible ejecutar la instrucción anterior dado que hay infinitos enteros positivos.

Ejemplo 2

Problema: Calcular la paga neta de un trabajador conociendo el número de horas trabajadas, la tarifa horaria y la tasa de impuestos.

Solución: Debemos definir el problema.

1. ¿Qué datos de entrada se requieren?
   Número de horas trabajadas
   Tarifa
   Impuestos
2. ¿Cuál es la salida deseada?
   Paga Neta

3. ¿Cuál es el método a usar? (Algoritmo)

Inicio

Leer Número de horas trabajadas

Leer Tarifa

Leer Impuestos

Calcular Paga Bruta = Número de horas trabajadas * Tarifa

Calcular Impuestos = Paga Bruta * Tasa

Calcular Pago Neta = Paga Bruta – Impuestos

Visualizar Paga Bruta

Visualizar Impuestos

Visualizar Pago Neta

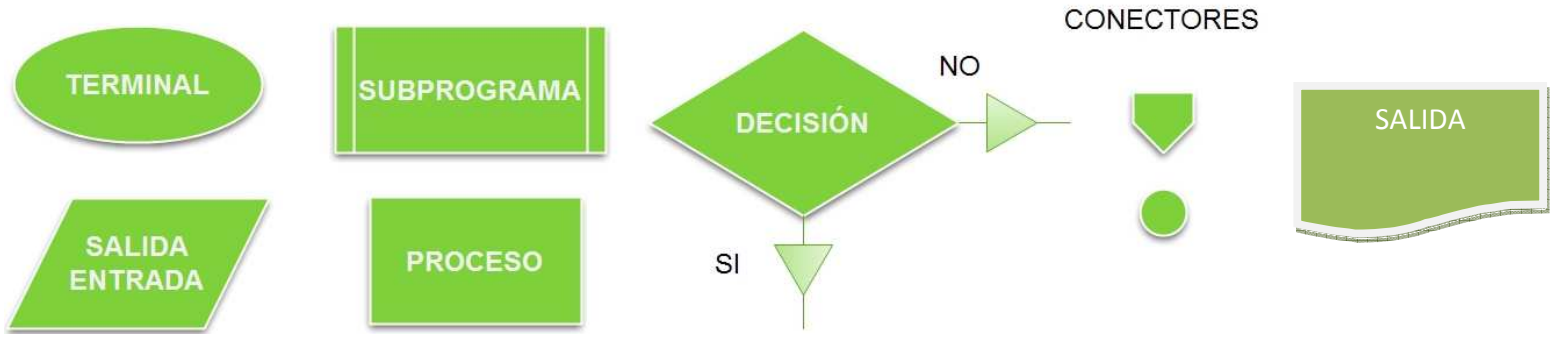
Fin

Diseño del Algoritmo

En esta fase, como se ha mencionado anteriormente, se determina cómo hace el programa la tarea solicitada. Los métodos más eficaces para el proceso de diseño se basan en el conocido divide y vencerás, esto es dividiendo el problema en subproblemas y a continuación dividir estos subproblemas en otros de nivel más bajo hasta que pueda ser implementada la solución.

Existen diferentes herramientas de programación, las más utilizadas para diseñar algoritmos son:

1. Diagramas de flujo: Es una representación gráfica de un algoritmo. Los símbolos normalizados por el Instituto Norteamericano de Normalización (ANSI) y los más frecuentes empleados se muestran a continuación.



2. Pseudocódigo: Es una herramienta de programación en la que las instrucciones se escriben en palabras similares en inglés o español, que facilitan tanto la escritura como la lectura de programas.

NIVEL 1 BÁSICO: ESTRUCTURAS DE CONTROL SELECTIVAS E ITERATIVAS

En sección se tratarán con las Estructuras de Control Selectivas e Iterativas, a continuación se describe brevemente cada una de dichas estructuras.

Estructuras de Control Selectivas:

Las estructuras selectivas se utilizan para tomar decisiones lógicas y existen en dos "formas": la sentencia **if-then-else** y la sentencia **switch**. La primera (**if-then-else**) se considera de alternativa doble (si se cumple cierta condición, entonces…, en caso contrario…), y tuene la siguiente estructura:

```
if (condición)
        acción 1;
else
        acción 2;
```

Nota: No se agrega ninguna llave "{" de apertura y cierre "}" cuando sólo es una instrucción. También no lleva la palabra reservada "then" en C.

También se puede dar el caso en que se tiene más instrucciones, esa opción deberá llevar una llave de apertura "{" y otra de cierre "}", por ejemplo:

```
if (condición)
 {
        Acción 1;
        Acción 2;
        Acción 3;
 }
else
 {
        Acción 4;
        Acción5;
 }
```

Nota: También puede darse el caso de que exista selecciones anidadas como se muestra a continuación.

```
if (condición1)
 {
        Acción 1;
        if (condición2)
                Acción 3;
 }
else
 {
        Acción 4;
        Acción5;
 }
```

Existe otra sentencia para la selección múltiple, esta es la sentencia **switch**, actúa como una función de bifurcación múltiple que sustituye con ventaja a **if-else-if** cuando el número de opciones que se puede presentar son numerosas. El mandato **switch**(variable) se utiliza para bifurcar las diferentes opciones presentadas según el valor ingresado para la variable en cada caso. Por ejemplo:

```
switch (variable)
{
        case constante1 : acción1;
             break;
        case constante1 : acción2;
             break;
        case constante1 : acción3;
             break;
        case constante1 : acción4;
             break;
        default: acción5;
}
```

Nota: Un ejemplo es donde evaluamos nota (tipo entero) y se imprime la calificación correspondiente al valor de nota, al final si se nota tiene 6 al 0, imprimirá en pantalla Reprobado.

```
switch (nota)
{
        case 10 : printf("Calificación 10");
             break;
        case 9 : printf("Calificación 9");
             break;
        case 8: printf("Calificación 8");
             break;
        case 7: printf("Calificación 7");
             break;
        default: printf("Reprobado");;
}
```

La sentencia **switch** se considera de selección múltiple, ya que el flujo de ejecución puede continuar por una cantidad N de alternativas posibles, según el valor de la expresión que se evalúa al principio.

Estructuras de Control Iterativas:

También conocidas como estructuras de repetición (bucles o ciclos) se utilizan para realizar varias veces el mismo conjunto de operaciones. Entre ellas se encuentran aquellas donde la cantidad de repeticiones se conoce a priori y aquellas en las que las repeticiones se realizan hasta que se cumple una condición lógica dada. En esta sección se verán las estructuras iterativas: ***for***, ***while*** y ***do-while***.

El primero es la estructura ***for***, permite definir un bucle controlado por un contador, denominado variable de control o de inducción, la sintaxis es:

```
for ( ...; ....; ....)
  {
        Acción1;
        Acción2;
        Acción3;
        Acción4;
        Acción5;
        ........
  }
```

Como se puede observar entre el paréntesis lleva "**;**", en la primera se inicializa la variable de control y sólo se ejecuta una vez. La segunda es la condición lógica que debe cumplirse, la tercera es la actualización de la variable de control.

```
acum = 0;
for (i=0; i<100; i++)
  {
        acum =acum +i;
        printf("%d ", acum);
        printf("\t  %d ", i);
        printf("\n");
  }
```

La estructura ***while***, por su parte, evalúa la condición lógica antes de comenzar cada iteración. Si ésta es verdadera, entonces se ejecuta el cuerpo de la estructura ***while***, en caso contrario, el bucle termina.

```
while (condición)
  {
        Acción1;
        Acción2;
        Acción3;
        Acción4;
        Acción5;
        ........
  }
```

Como se puede observar en el siguiente código, la variable de control "i" se inicializa antes, posteriormente es la que se compara en la condición, por ello se debe actualizar dentro del bucle (i=i+1 puede sustituirse por una forma abreviada i++;).

```
i=0;
acum = 0;
while (i < 100)
  {
        acum = acum +i;
        printf("%d ", acum);
        printf("\t  %d ", i);
        printf("\n");
        i= i+1;
  }
```

Por último el bucle ***do-while***, se utiliza cuando se quiere asegurar que el ciclo se ejecuta al menos una vez, puesto que la evaluación de la condición lógica se hace al final de éste.

```
do
  {
        Acción1;
        Acción2;
        Acción3;
        Acción4;
        Acción5;
        ……..
  } while (condición);
```

En este ejemplo se ejecuta primero (sólo una vez) y posteriormente verifica la condición si es verdadera, en caso contrario se sale del bucle.

```
i=0;
acum = 0;
do
  {
        acum = acum +i;
        printf("%d ", acum);
        printf("\t  %d ", i);
        printf("\n");
        i= i+1;
  } while (i < 100)
```

A continuación se listan ejercicios resueltos de esta sección.

Ejercicios Resueltos

Problema 1: Diseñe un algoritmo que, dado un número real que entra como dato, nos indique si está contenido dentro de los límites predeterminados. El límite inferior es de 100 y el superior de 200.

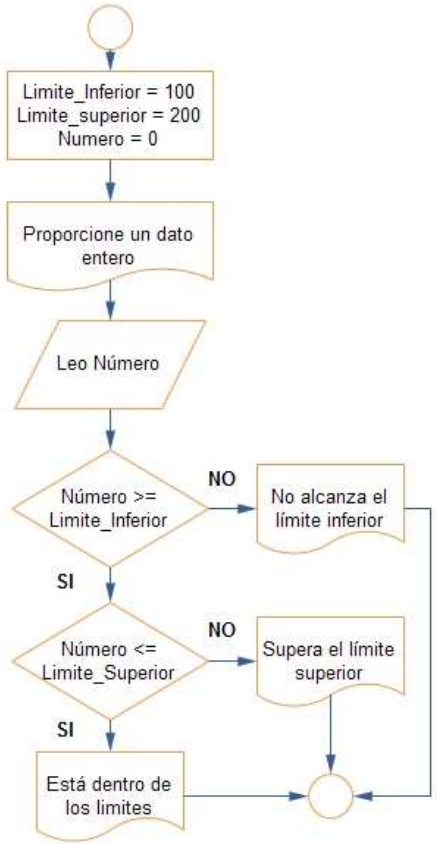
Restricciones: Uso de las estructuras de selección if-then-else.

Algoritmo

Datos de Entrada:
Número de tipo real
Datos de Salida:
Mensaje de que está dentro de los límites, mensaje de que está fuera de rango o no alcanza
Algoritmo:

```
Inicio
    Limite_Inferior = 100
    Limite_Superior = 200
    Solicitar número al usuario.
    Almaceno en mi variable Número
    Si Número es mayor o igual que Limite_Inferior entonces
        Si Número es menor o igual que Limite_superior entonces
            Imprimo en pantalla que está dentro de los límites
        Sino
            Imprimo en pantalla que supera al límite máximo
    Sino
        Imprimo en pantalla que no alcanza el límite mínimo
fin
```

Diagrama de Flujo o Pseudocódigo



Código en C

```c
#include <stdio.h>
#include <conio.h>
#define Limite_Inferior 100
#define Limite_Superior 200

int main()
{
  float Numero=0; //Definimos nuestra variable
  printf("----Problema 1-----\n");
  printf("Introduzca un número: ");
  scanf("%f", &Numero);
  if (Numero >= Limite_Inferior)
    {
      if (Numero <= Limite_Superior)
        {
          printf("Está dentro del intervalo");
        }
      else
        printf("Supera el límite máximo ");
    }
  else
    printf("No alcanza el límite mínimo");

  return 0;
}
```

Problema 2: Programe un algoritmo que, dados dos números enteros que entran como datos, indique si uno es divisor del otro.

Restricciones: Uso de las estructuras de selección if-then-else.

Algoritmo

Datos de Entrada:

Número1 y Número2 de tipo entero

Datos de Salida:

Mensaje de que Número1 es divisor de de Número2 o que Número2 es divisor de Número1

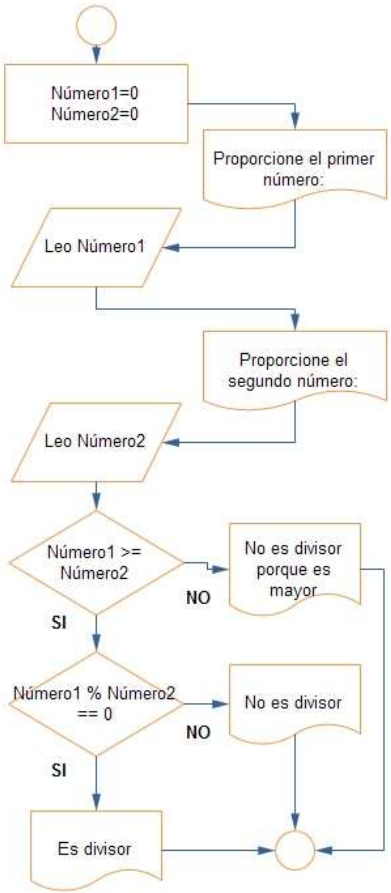
Algoritmo:

Inicio

Solicitar el primer número al usuario.
Almaceno en el Número1
Solicitar el segundo número al usuario.
Almaceno en el Número2
Si Número1 es mayor que Número2 entonces
Si Número1 módulo Número2 es igual a 0 entonces
Imprimo en pantalla Número2 es divisor de Número1
Sino
Imprimo en pantalla Número2 no es divisor de Número1
Sino
Imprimo en pantalla no es divisor porque Número2 es mayor que Número1

fin

Diagrama de Flujo o Pseudocódigo



Código en C

```
#include <stdio.h>
#include <conio.h>

int main()
{
  int Numero1=0, Numero2=0; //Variables
  printf("----Problema 2-----\n");
  printf("Introduzca el primer número: ");
  scanf("%i", &Numero1); //indica sin signo
  printf("Introduzca el segundo número: ");
  scanf("%i", &Numero2); //indica sin signo
  if (Numero1 > Numero2)
    {
      if (Numero1%Numero2 == 0)
        {
          printf("Es divisor %i de %i", Numero1,
          Numero2);
        }
      else
        printf("%i no es divisor de %i ", Numero2,
        Numero1);
    }
  else
    printf("No es divisor porque es mayor %i",
    Numero2);

  return 0;
}
```

Problema 3: Un triangulo rectángulo puede tener lados que sean todos enteros. El conjunto de tres valores enteros para los lados de un triángulo rectángulo se conoce como una terna pitagórica. Estos tres lados deben satisfacer la relación de que la suma de los cuadrados de dos lados es igual al cuadrado de la hipotenusa. Encuentre todas las ternas de Pitágoras para el cateto opuesto, cateto adyacente e hipotenusa, todos ellos no mayores de 500.

Restricciones: Uso de las estructuras de selección if-then-else y de estructuras repetitivas.

Algoritmo

Datos de Entrada:

Los valores del 1 al 500 de tipo entero.

Datos de Salida:

Mensaje con los valores de las ternas pitagóricas.

Mensaje con los valores de hipotenusa, opuesto y adyacente elevados al cuadrado, como comprobación

Algoritmo:

Inicio

Imprimo mensaje Ternas Pitagoricas

Para i=1, hasta i menor o igual a 500, incrementa i en uno

hipotenusa=i*i

j=1

Para j=1, hasta j menor o igual a 500, incrementa i en uno

opuesto= j*j

Para k=1, hasta k menor o igual a 500, incrementa i en uno

adyacente=k*k

Suma de Cuadrados = opuesto + adyacente

Si Suma de Cuadrados es igual a hipotenusa entonces

Imprimo mensaje con los valores i, j y k

Imprimo mensaje con los valores de opuesto, adyacente e hipotenusa

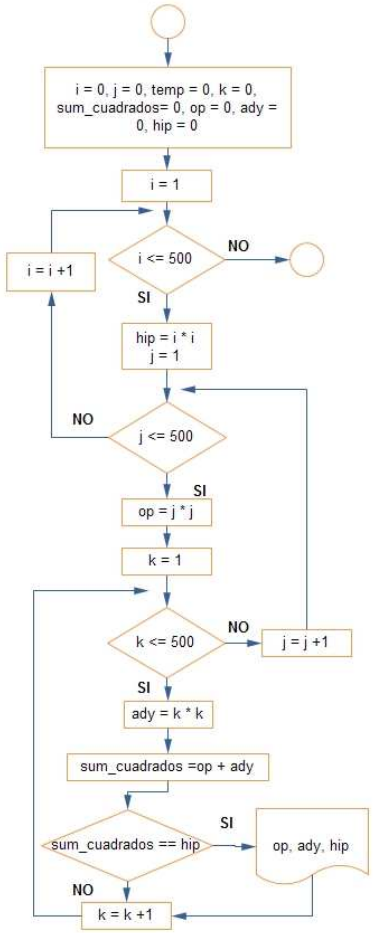
Fin de ciclo k

Fin de ciclo j

Fin de ciclo i

Fin

Diagrama de Flujo o Pseudocódigo



Código en C

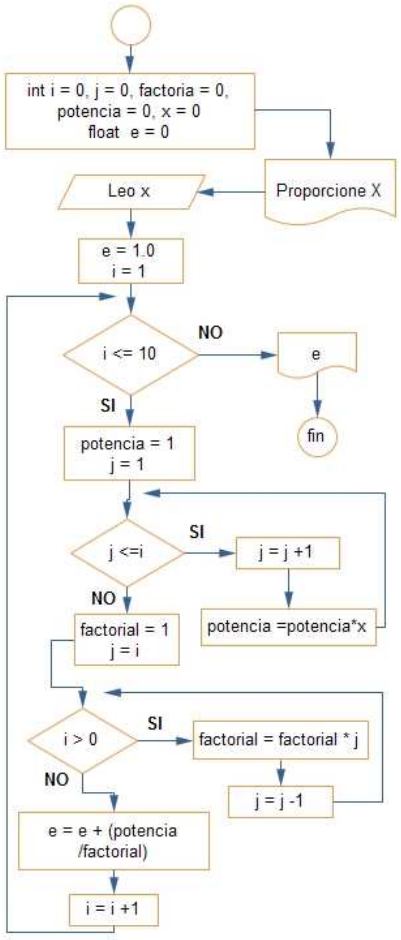
```
#include <stdio.h>
#include <windows.h>

int main()
{
   int i,j,k,Sum_Cuadrados,op,ady,hip;
   printf("Ternas Pitagoricas\n");
   for (i=1;i<=500;i++)
   {
      hip=i*i;// eleva la hipotenusa al cuadrado
      for (j=1;j<=500;j++)
      {
         op=j*j;//eleva el cateo opuesto al cuadrado
         for (k=1;k<=500;k++)
          {
            ady=k*k;
            Sum_Cuadrados=op+ady;
            if (Sum_Cuadrados==hip)//determina si existe
                                   // una terna pitagórica
            {
            printf("\n cateto opuesto= %d,",j);
            printf(" cateto adyacente= %d,",k);
            printf(" cateto hipotenusa= %d",i);
            printf("\n Comprobacion: %d + %d = %d
\n",op,ady,hip);
            }
          }
      }
   }
   Sleep(10000);
   return 0;
}
```

Problema 4: Escriba un programa que calcule el valor de e^x
Recuerde que dicha constante matemática se calcula de la siguiente forma: $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$
Y el factorial de todo número se obtiene así: $n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 1$

Restricciones: Uso de las estructuras de selección if-then-else y de estructuras repetitivas.

Algoritmo	Diagrama de Flujo o Pseudocódigo	Código en C

Datos de Entrada:
x de tipo entero.
Datos de Salida:
Mensaje de con el valor de e^x
Algoritmo:
Inicio
Solicitar el valor x.
Almaceno en x
e=1.0
Para i=1, hasta i menor o igual a 10, incrementa i en uno
Potencia=1
Para j=1, hasta j menor o igual a i, incrementa i en uno
Potencia=potencia*x
Fin de ciclo j
Para j=i, hasta j mayor a 0, decrementa i en uno
Factorial=Factorial*i
Fin de ciclo j
e=e+(potencia/factorial);
Fin del ciclo i
Imprimo mensaje con el valor de e
Fin




```c
#include <stdio.h>
#include <windows.h>

int main()
{
   int i,j,factorial,potencia,x;
   float e;
   printf("Introduce el valor de x: ");
   scanf("%d",&x);
      // Calcula los primero 10 valores para e elevado a x.
   e=1.0;
   for (i=1;i<=10;i++)
   {
         //Calcula  x elevada a una potencia i
      potencia=1;
      for (j=1;j<=i;j++)
            potencia=potencia*x;
         // Calcula el Factorial de i
      factorial=1;
      for (j=i;j>0;j--)
         factorial=factorial*j;
         //Acumula el valor de e
      e=e+(potencia/factorial);
   }
   printf("El valor de e elevado a x: %f", e);
   Sleep(10000);
   return 0;
}
```

Ejercicios Propuestos

Problema 1: Escribir un programa que, dado un número real cualquiera, encuentre su valor absoluto. El valor absoluto de un número x es igual a x si x > 0, y a -x si x es menor o igual a 0. Por ejemplo, el valor absoluto de 0.5 es 0.5, mientras que el valor absoluto de 3 es 3.
Restricciones: Debe utilizar una estructura de control si-entonces-sino.
Problema 2: En un videoclub se ofrece la promoción de llevarse tres películas por el precio de las dos más baratas. Haga un programa que, dados los tres precios de las películas, determine la cantidad a pagar.
Restricciones: Debe utilizar una estructura de control si-entonces-sino.
Problema 3: Escriba un programa que reciba cuatro calificaciones de un estudiante y devuelva el promedio y la máxima y la mínima calificación.
Restricciones: Debe utilizar estructuras de selectivas e iterativas.
Problema 4: Considere siguiente proceso repetitivo para un número entero dado: si el número es 1, el proceso termina. De lo contrario, si es par se divide entre 2, y si es impar se multiplica por 3 y se le suma 1. Si empezamos con 6, por ejemplo, obtendremos la sucesión de números 6, 3, 10, 5, 16, 8, 4, 2, 1. La conjetura de Collatz dice que, a partir de cualquier número inicial, la sucesión obtenida siempre termina en 1. Escriba un programa que permita verificar la conjetura de Collatz para cualquier entero dado, y que imprima la secuencia correspondiente.
Restricciones: Debe utilizar estructuras de selectivas e iterativas.

NIVEL 2 MEDIO: FUNCIONES Y RECURSIVIDAD

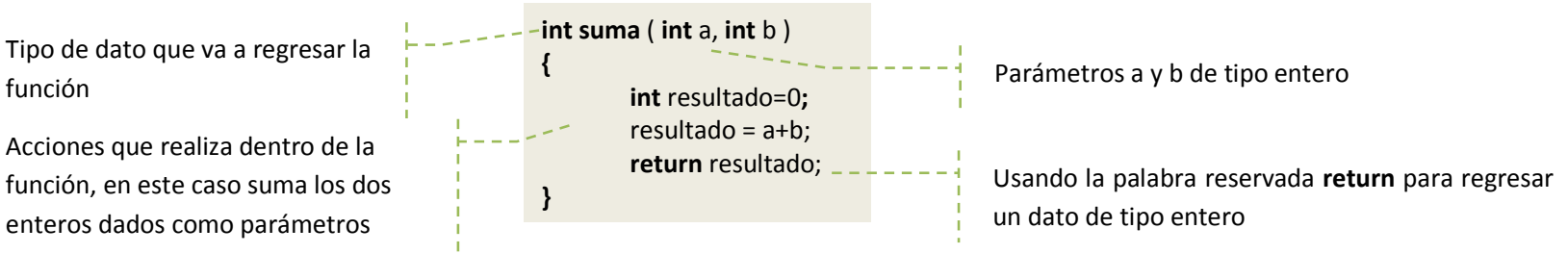
En esta sección se tratarán con las funciones y recursividad.

Funciones: En lenguaje C, la modularización se lleva a cabo mediante el uso de funciones, que luego cooperarán entre sí para realizar las tareas necesarias de manera de obtener el resultado deseado por el usuario final. Como se puede observar el ejemplo siguiente:

```
<tipo de retorno> <nombre de la función> ( <lista de parámetros> )
{
        <acciones que realiza la función>
}
```

Nota: la parte encerrada entre llaves "{"y "}" se le denomina cuerpo, mientras que el encabezado integrado por el nombre de la función, el tipo de retorno y su lista de parámetros se le conoce como interfaz o prototipo

Una función puede ser llamada desde cualquier parte del programa que la contiene, posterior a su declaración/definición. Se le invoca con su nombre, seguido de una lista opcional de argumentos (o parámetros). Los argumentos van entre paréntesis y, si hubiera más de uno, separados por comas. Cuando la función no devuelve valor alguno, se reemplaza el <tipo de retorno> por la palabra reservada ***void***. Por Ejemplo:



En el ejemplo anterior la función se llama suma, recibe dos parámetros de tipo entero (a y b), y retorna un entero (que es la suma de los parámetros). Una función siempre retorna un solo valor (en casos eventuales no podría no devolver nada si se usa ***void*** como tipo de retorno). Si fuese necesario de devolver más de un valor,

debería realizarse por medio de los parámetros, que dejarían de ser sólo de entrada, para convertirse en parámetros de entrada y salida.

A continuación se describe la declaración/definición así como la invocación del a función.

```
#include <stdio.h>
#include <conio.h>

int suma(int a, int b);

int main()
{
    int Operando1, Operando2, resultado;
    Operando1 = 5;
    Operando2 =10;
    resultado = suma(Operando1, Operando2);
    printf("%d + %d = %d \n", Operando1, Operando2, resultado);
    getch();
    return 0;
}

int suma(int a, int b)
{
    int resultado_suma=0;
    resultado_suma = a+b;
    return resultado_suma;
}
```

Declaración de la función, en esta parte se define que tipo de valor va a regresar la función y los parámetros (constituyen la vía a través de la cual la rutina interactúa con el exterior, intercambiando datos) que necesita para realizar dicha función.

Asignación de valores a las variables locales Operando1 y Operando2, aunque se pueden solicitar al usuario.

Invocación de la función antes declarada, con los parámetros que solicita, la función suma requiere dos elementos para que funcione.

Definición de la función (recuerde que en esta parte no se le pone ";" al final, el parámetro es el nombre de la variable que utiliza la función en forma interna para hacer uso de cada valor que le está pasando el que la llamó.

La variable resultado_suma es una variable local, por lo cual no es reconocida en el programa principal sólo en ésta función.

Recursividad: Se presenta cuando una función se invoca a si misma. Distintamente a las iteraciones (bucles), las funciones recursivas consumen muchos recursos de memoria y tiempo.

Una función recursiva se programa simplemente para resolver los casos más sencillos, cuando se llama a una función con un caso más complicado, se divide el problema en dos partes, la parte que se resuelve

inmediatamente y la que necesita de más pasos, ésta última se manda de nuevo a la función, que a su vez la divide de nuevo, y así sucesivamente hasta que se llegue el caso base. Cuando se llegue al final de la serie de llamadas, va recorriendo el camino de regreso, hasta que por fin, presenta el resultado.

La recursividad es un tema importante en el mundo de la programación, la utilidad de este tipo de funciones se aprovecha en el diseño de algoritmos de criptografía, de búsqueda, entre otros. La implementación de la recursividad en nuestros programas debe evaluarse con mucho cuidado, debemos tratar de evitar que se usen funciones de complejidad exponencial, que se llamen a sí mismas una y otra vez sin que se tenga un control claro.

Un ejemplo clásico para este problema es el factorial de un número, (n!). Conociendo los casos base 0! = 1 y 1!=1, se sabe que la función factorial puede definirse como n!=n*(n-1)!, entonces 5!=5*4! que a su vez 5*4*3! que a su vez es 5*4*3*2! entonces 5*4*3*2*1! y por último 5*4*3*2*1*0! llegando al caso base. Por ejemplo:

Declaración de la función recursiva.

Invocación de la función recursiva.

Definición de la función recursiva.

Invocación de sí misma pero con n-1.

```
#include <stdio.h>
#include <conio.h>

float factorial(float numero);

int main()
{
    float n=0;
    printf("Introduzca un número para calcular el factorial \n");
    scanf("%f", &n);
    printf("\n El factorial es: %f", factorial(n));
    getch();
    return 0;
}

float factorial(float numero)
{
    If (numero <= 1)
        return 1;
    else
        return numero * (factorial( numero-1));;
}
```

A continuación se presentan ejercicios resueltos, para dichos ejercicios se recomienda realizar pruebas de escritorio.

Ejercicios Resueltos

Problema 1: El juego de dados conocido como "craps" (tiro perdedor) es muy popular, realice un programa que simule dicho juego, a continuación se muestran las reglas para los jugadores.

- Un jugador tira dos dados. Cada dato tiene seis caras. Las caras contienen 1, 2, 3, 4, 5 y 6 puntos.
- Una vez que los dados se hayan detenido, se calcula la suma de los puntos en las dos caras superiores.
- Si a la primera tirada, la suma es 7, o bien 11, el jugador gana.
- Si a la primera tirada la suma es 2, 3 o 12 (conocido como "craps"), el jugador pierde (es decir la casa "gana").
- Si a la primera tirada la suma es 4, 5, 6, 8, 9 ó 10, entonces dicha suma se convierte en el "punto" o en la "tirada".
- Para ganar, el jugador deberá continuar tirando los dados hasta que haga su "tirada".
- El jugador perderá si antes de hacer su tirada sale una tirada de 7.

Restricciones: Use funciones para este problema, así como las estructuras de control selectivas y repetitivas.

Algoritmo

Datos de Entrada:

Puntos aleatorios de los dados generados por el programa.

Datos de Salida:

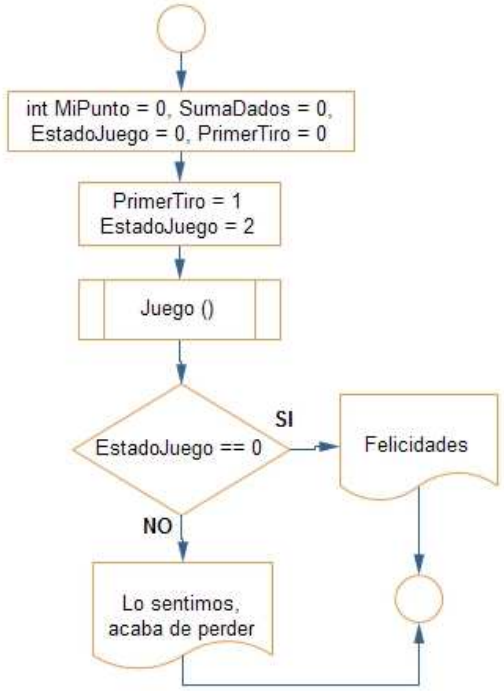
Mensaje con el resultado del juego. "Felicidades" si se ha ganado el juego, "Lo sentimos acaba de perder" si se ha perdido la partida.

Algoritmo:

Inicio

EstadoJuego=Continua
Mientras EstadoJuego=Continua
Se lanzan los dados
  Si es el primer lanzamiento entonces
    Si el lanzamiento fue igual a 7 u 11 entonces
      EstadoJuego=GANA
      Imprimo los puntos del lanzamiento.
    Si el lanzamiento fue igual a 2, 3 o 12 entonces
      EstadoJuego=PIERDE
      Imprimo los puntos del lanzamiento.

Diagrama de Flujo o Pseudocódigo

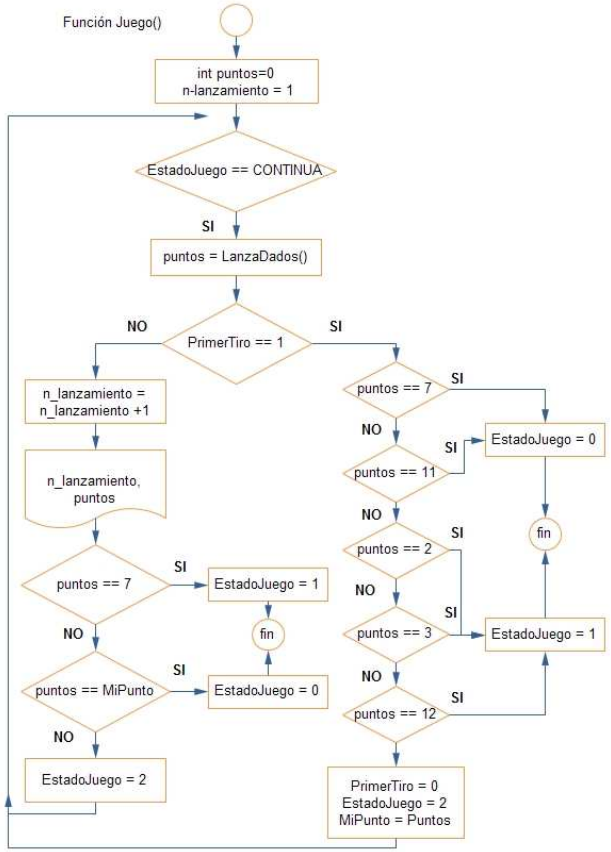


Código en C

```
#include <stdio.h>
#include <windows.h>

// varibles utilizadas para comprobar el estado del juego
#define GANA 0
#define PIERDE 1
#define CONTINUA 2

int PrimerTiro=1, // 1 si el lanzamiento actual es el primero
   SumaDados=0,  // suma de los dados
   MiPunto=0,  // tirada o puntos si no gana o pierde en el primer lanzamiento
   EstadoJuego=CONTINUA;

int LanzaDados();
int Juego();

int main()
{
   int i;
   printf("\n CRAPS \n");
   Juego();
   if (EstadoJuego==GANA)
```

Si el lanzamiento fue igual a 4, 5, 6, 8, 9 o 10.
EstadoJuego=CONTINUA
Imprimo los puntos del lanzamiento.
Puntos=Primer_lanzamiento
Sino
Si el lanzamientos fue igual a Puntos
EstadoJuego=GANA
Imprimo los puntos del lanzamiento
Si el lanzamiento fue igual a 7
EstadoJuego=PIERDE
Imprimo los puntos del lanzamiento
Si el lanzamiento no fue igual a 7 o no fue igual a Puntos
EstadoJuego=CONTINUA
Imprimo los puntos del lanzamiento
Fin del ciclo Mientras

Fin




```
    printf("\n ¡FELICIDADES! ");
  else
    printf("\n Lo sentimos acaba de perder ");
  Sleep(5000);
 return 0;
}

int LanzaDados()
{
  int dado1,dado2, suma;
  dado1=1+(rand()%6);
  dado2=1+(rand()%6);
  suma=dado1+dado2;
  return suma;
}

int Juego()
{
  int puntos,n_lanzamiento;
  n_lanzamiento=1;
  while(EstadoJuego==CONTINUA)
  {
  puntos=LanzaDados();
  if (PrimerTiro==1)
  {
    printf("\n Primer lanzamiento: %d", puntos);
    switch(puntos)
        {
        case 7:EstadoJuego=GANA;break;
        case 11:EstadoJuego=GANA;break;
        case 2:EstadoJuego=PIERDE;break;
        case 3:EstadoJuego=PIERDE;break;
        case 12:EstadoJuego=PIERDE;break;
        default: {
                PrimerTiro=0;
                EstadoJuego=CONTINUA;
                MiPunto=puntos;
                break;
                }
        }
  }
  else
  {
```

		n_lanzamiento=n_lanzamiento+1;
		printf("\n Lanzamiento numero %d: %d",
		n_lanzamiento,puntos);
		switch(puntos)
		{
		case 7:EstadoJuego=PIERDE;break;
		case 'Mipunto':EstadoJuego=GANA;break;
		default: {
		PrimerTiro=0;
		EstadoJuego=CONTINUA;
		MiPunto=puntos;
		break;
		}
		}
		}
		}
		return EstadoJuego;
		}

Problema 2: Realice un programa que realice las funciones que a continuación se indican. Todas las opciones deben ser presentadas al usuario a través de un menú de opciones.

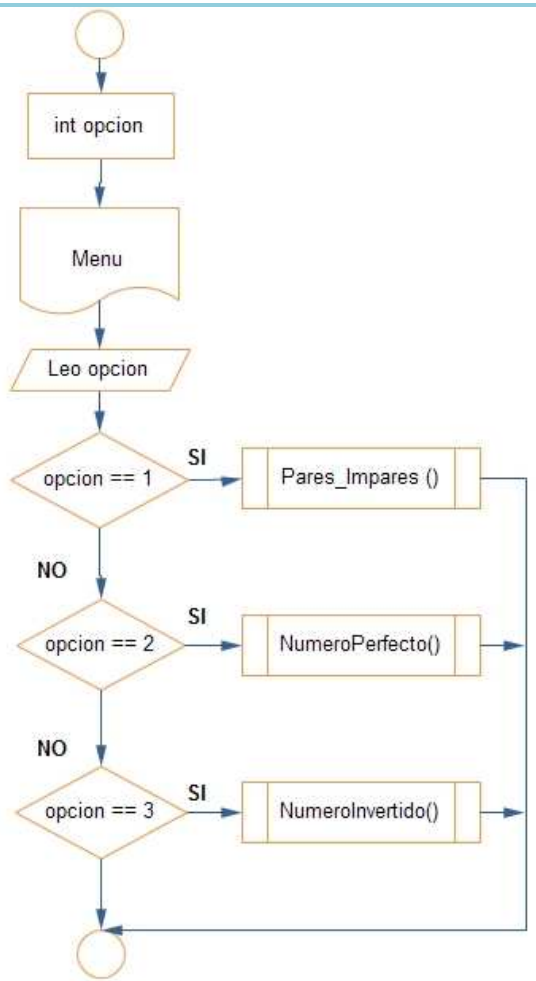
- Escriba una función en la que se introduzcan 10 enteros y determine cuales de estos enteros son pares y cuales son impares.
- Un número entero es un "número perfecto" si sus factores, incluyendo al 1 (pero excluyendo en el número mismo), suman igual que el numero. Ejemplo: 6 es un numero perfecto porque 6= 1+2+3. Escriba una función que regrese los primeros 100 números perfectos. Esta función debe tener una función anidada que determine al número perfecto.
- Escriba una función que tome un valor entero de cuatro dígitos y regrese el número con los dígitos invertidos. Por ejemplo, dado el número 7631, la función deberá regresar 1367.

Restricciones: Use funciones para este problema, así como las estructuras de control selectivas y repetitivas.

Algoritmo	Diagrama de Flujo o Pseudocódigo	Código en C

Datos de Entrada:
Opción del menú de tipo entero.
10 números enteros para la función de pares e impares.
100 primeros números enteros para la función del número perfecto.
Un número de cuatro dígitos, para la función que invierte el número.
Datos de Salida:
Mensajes para números impares y pares.
Los números perfectos del 1 al 100.
Mensaje con los dígitos invertidos.
Algoritmo:
Inicio
Imprime la opción "1.- Pares es impares"
Imprime la opción "2.-Numero Perfecto"
Imprime la opción "3.-Numero Invertido"
Lee la opción
Si la opción es 1 entonces
Llama a la función para números pares e impares
Si la opción es 2 entonces
Llama a la función para determinar los números perfectos del 1 al 100
Si la opción es 3 entonces
Llama a la función que invierte un número de cuatro dígitos
	(En la siguiente página se presenta el DF)	#include <stdio.h>
		#include <windows.h>
		
		// Prototipos de la funciones del programa
		void Pares_Impares();
		int Perfecto(int n);
		void Numero_Perfecto();
		void Inverso();
		void Tiempo();
		
		int main()
		{
		int opcion;
		printf("MENU DE OPCIONES \n");
		printf("\n 1.- Pares e Impares ");
		printf("\n 2.- Numero Perfecto ");
		printf("\n 3.- Numero Invertido ");
		printf("\n Seleccione una opcion: ");
		scanf("%d",&opcion);
		switch(opcion)
		{
		case 1: Pares_Impares(); break;
		case 2: Numero_Perfecto(); break;
		case 3: Inverso(); break;
		default: break;
		}
		Sleep(5000);
		return 0;
		}

Fin




```
//Función que determina los números pares e impares.
void Pares_Impares()
{
    int i=0, v, t;
    printf("\nNUMEROS PARES E IMPARES \n");
    for(i=1;i<=10;i++)
     {
       printf("\n Introduce el valor %d: ",i);
       scanf("%d",&v);
       t=v%2;
       if (t==0)
         printf(" Es un valor par \n");
       else
         printf(" Es un valor inpar \n");
     }
}

//Funcion que determina si un número es perfecto.
int Perfecto(int n)
{
    int i, suma,t;
    suma=0;
    for (i=1;i<n;i++)
    {
       t=n%i;
       if (t==0)
         suma=suma+i;
    }
    if  (suma==n)
       return 1;
    else
       return 0;
}

   //Funcion que encuentra los numeros perfectos del 1
   al 100.
void Numero_Perfecto()
{
    int i,t;
    printf("\nNUMEROS PERFECTOS \n");
    for (i=1;i<=100;i++)
    {
```

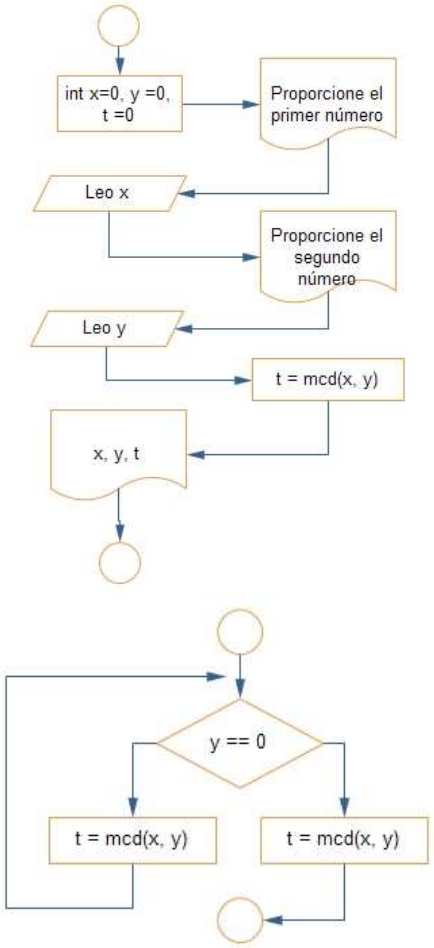
```
      t=Perfecto(i);
      if (t==1)
        printf(" %d Es un numero perfecto \n",i);
    }
}

//Fucion que invierte el orden de los digitos de un numero.
void Inverso()
{
   int i,x,t, suma;
   printf("\n NUMERO INVERTIDO \n");
   printf("\n Ingrese el numero de 4 digitos: ");
   scanf("%d",&x);
   suma=0;
   t=x/1000;
   x=x%1000;
   suma=suma+t;
   t=x/100;
   x=x%100;
   suma=suma+(t*10);
   t=x/10;
   x=x%10;
   suma=suma+(t*100);
   suma=suma+(x*1000);
   printf("\n El numero es=%d ", suma);
}
```

Problema 3: Escriba una función recursiva MCD que regrese el máximo común divisor de *x y* de *y*.
El máximo común divisor de los enteros *x* e *y* es el número más grande que divide en forma completa tanto a *x* como a *y*.
El *gcd* de *x y* de *y*, se define en forma recursiva como sigue: Si y es igual a 0, entonces gcd (de *x, y*) es *x*; de lo contrario *gcd (de x, y)* es igual a *gcd(y, x%y).*

Restricciones: Use recursividad para este problema, así como las estructuras de control selectivas y repetitivas.

Algoritmo	Diagrama de Flujo o Pseudocódigo	Código en C

Datos de Entrada:
El valor de x, el cual es numero entero.
El valor de y, el cual es numero entero.
Datos de Salida:
El máximo común divisor de x, y, el cual es número entero.
Algoritmo:
Inicio
Lee el valor de de x
Lee el valor de de y
Si y es igual a 0 entonces
Regresa el valor de x
Sino
Regresa la función mcd(y,x%y)
Fin




```c
#include <stdio.h>
#include <windows.h>

   // Prototipos de la funciones del programa
int mcd(int x, int y);

int main()
{
   int x,y,t;
   printf("\n MAXIMO COMUN DIVISOR\n");
   printf("\n Ingrese el valor de x:  ");
   scanf("%d",&x);
   printf("\n Ingrese el valor de y:  ");
   scanf("%d",&y);
   t=mcd(x,y);
   printf("\n El MCD de x=%d y y=%d es: %d ",x,y,t);
 Sleep(5000);
 return 0;
}

int mcd(int x, int y)
{
      if(y==0)
            return x;
      else
            //Recursividad
         return mcd(y,x%y);
}
```

Problema 4: En teoría de la computación, la función de Ackermann es una función recursiva que toma dos números naturales como argumentos y devuelve un único número natural. Escriba un programa en C, que determine la serie de Ackerman cuando se proporcionan los parámetros *m* y *n*. Como norma general se define como sigue:

$$A(m,n) = \begin{cases} n+1 & Si\ m = 0 \\ A(m-1,1) & Si\ m > 0\ \ y\ \ n = 0 \\ A(m-1, A(m,n-1)) & Si\ m > 0\ \ y\ \ n > 0 \end{cases}$$

Ejemplo:

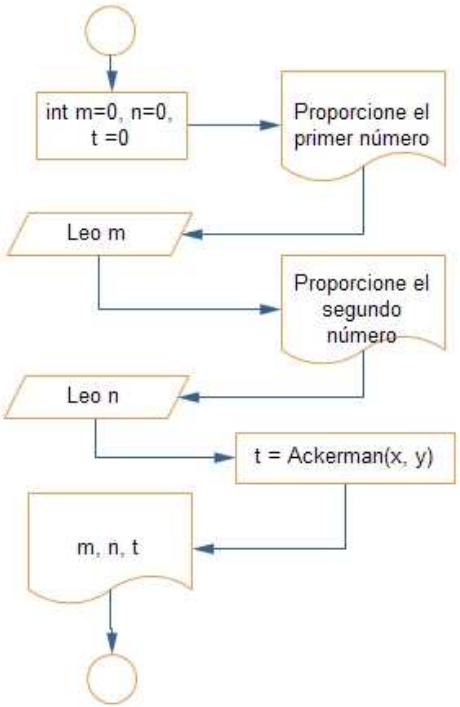
La primera fila de la función de Ackerman contiene los enteros positivos, dado que A(0, n)) consiste en sumar uno a n. El resto de las filas se pueden ver como indirecciones hacia la primera. En el caso de m = 1, se redirecciona hacia A(0, n + 1), sin embargo, la simplificación es algo complicada.

Restricciones: Use recursividad para este problema, así como las estructuras de control selectivas y repetitivas.

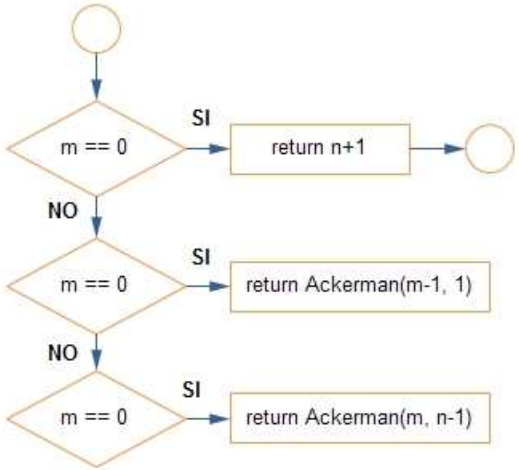
Algoritmo	Diagrama de Flujo o Pseudocódigo	Código en C

Datos de Entrada:
El valor de m, el cual es numero entero.
El valor de n, el cual es numero entero.
Datos de Salida:
La función de Ackerman para m y n
Algoritmo:
Inicio
Lee el valor de de m
Lee el valor de de n
Si m es igual a 0 entonces
Regresa n+1
Si m es mayor a 0, y n es igual a 0 entonces
Regresa Ackerman(m-1,1)
Si m es mayor a 0, y n es mayor a 0 entonces
Regresa Ackerman(m-1,Ackerman(m,n-1))

Fin




```
#include <stdio.h>
#include <windows.h>

// Prototipos de la funciones del programa
int ackerman(int m, int n);

int main()
{
   int m,n,t;
   printf("\n FUNCION DE ACKERMAN\n");
   printf("\n Ingrese el valor de m:  ");
   scanf("%d",&m);
   printf("\n Ingrese el valor de n:  ");
   scanf("%d",&n);
   t=ackerman(m,n);
   printf("\n La funcion de Ackerman para m=%d y n=%d
es: %d ",m,n,t);
   Sleep(5000);
   return 0;
}

int ackerman(int m, int n)
{
   if(m==0)
         return n+1;
   else
```




```
    //Parte Recursiva
    {
      if(n==0)
            return ackerman(m-1, 1);
      else
            return ackerman(m-1, ackerman(m, n-1));
    }
}
```

Ejercicios Propuestos

Problema 1: Evalué las siguientes expresiones donde i y n son proporcionadas por el usuario, proponga un menú para cada caso.

- $\sum_{i=1}^{n} \sqrt{x2 - 4}$
- $\sum_{i=1}^{n} \frac{2^{i+1} - 2^{i}}{i+1}$
- $\sum_{i=1}^{n} (-1)^{i+1} \frac{i}{2^{i}}$
- $\prod_{i=1}^{n} n(n+1)$

Restricciones: Uso de funciones así como las estructuras de control selectivas y repetitivas.

Problema 2: Implemente un programa que calcule las siguientes integrales definidas, dado [a, b].

- $\int_{a}^{b} \frac{x}{x^2+3} dx$
- $\int_{a}^{b} \frac{dx}{1+x^2}$
- $\int_{a}^{b} senx\ dx$
- $\int_{a}^{b} senx\ cosx\ dx$

Restricciones: Uso de funciones así como las estructuras de control selectivas y repetitivas.

Problema 3: Escriba un programa que despliegue todas las permutaciones de los números del 1 a n. Por ejemplo, si n = 3, debe obtenerse:

1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

Restricciones: Debe tener funciones recursivas.

Problema 4: Escriba un programa que despliegue todos los subconjuntos no vacíos del conjunto de los números del 1 a n. Por ejemplo, si n = 3, debe obtenerse:

1
2
3
1 2
1 3
2 3
1 2 3

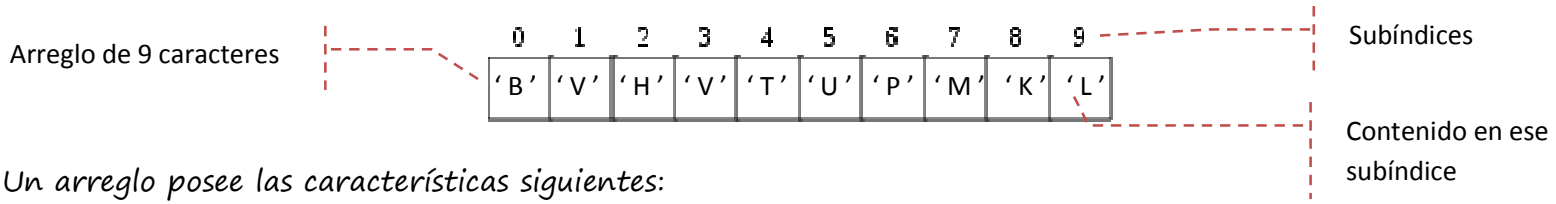
Restricciones: Debe tener funciones recursivas.

NIVEL 3 AVANZADO: ARREGLOS Y PUNTEROS

En esta sección se tratarán con arreglos y punteros.

Arreglos: Los ejemplos anteriormente presentados trabajan con datos simples, como números (enteros y reales) y caracteres. Sin embargo, en la práctica suele ser necesario procesar conjunto de datos, como podrían ser lista de clientes, cuentas bancarias, alumnos, entre otros. Por ejemplo, ¿cómo se podría implementar un algoritmo que recorra un listado de clientes de una compañía para calcular el monto total de adeudado por todos ellos? Está claro que sería necesario utilizar una estructura de control repetitiva que itere sobre cada cliente, consultando su deuda y acumulándola en una variable. No obstante, ¿cómo mantenemos los datos de, por ejemplo, 5 000 clientes en un programa?¿Significa que deben declararse 5 000 variables, una para cada cliente? Es necesario que el lenguaje de programación brinde soporte para mantener conjuntos de datos (no tan sólo variables simples como las secciones anteriores). Este soporte está dado por lo que se conoce como vectores (arreglos lineales) y matrices (arreglos multidimensionales), y que se tratará a continuación.

Un arreglo lineal (también conocido como vector) es un tipo de dato que se construye a partir de elementos más simples, cada uno de ellos identificado con un índice, por ejemplo:



Un arreglo posee las características siguientes:

Es una estructura homogénea: Esto quiere decir que todos los datos almacenados en esa estructura son del mismo tipo (por ejemplo, todos de tipo entero o todos de tipo carácter, o todos reales, entre otros).

Es una estructura lineal de acceso directo: Esto significa que se puede acceder a los datos en forma directa, con sólo proporcionar su posición (en forma de subíndice).

Es una estructura estática: Su tamaño (número y tipo de elementos) se define en tiempo de compilación y no cambia durante la ejecución del programa.

Un vector se declara indicando el tipo de dato base y la cantidad de elementos almacenados, según la sintaxis siguiente:

```
<tipo base>  <nombre de la variable> [ <cantidad de elementos>];
```

En C los arreglos se indexan desde la posición 0 hasta la posición N-1 inclusive siendo N el tamaño del arreglo.

Ejemplos:

```
char arreglo_letras [ 10]; //Arreglo de letras de tamaño 10
int arrreglo_enteros [30]; //Arreglo de enteros de tamaño 30
float arrreglo_num_reales [20]; //Arreglo de reales de tamaño 20
```

Se puede acceder a cada una de las posiciones tanto para lectura como para escritura, y para esto es necesario indicar el subíndice, veamos un ejemplo de ello:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>
#define N 10

int main ()
{
  int enteros[N];
  srand(time(NULL));

  for (i=0; i<N; i++)
    {
      enteros[i] = rand();
    }
}
```

Declaración de librerías que se van a utilizar. Se le recomienda al lector revisar las funciones que contienen dichas librerías.

Se define el tamaño y la variable con el valor de 10.

Se declara nuestro arreglo de tipo entero.

Se usa el ciclo iterativo para almacenar los enteros en el arreglo

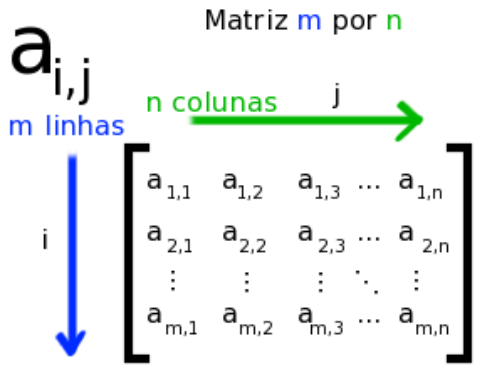
Se usa la función rand que toma valores aleatorios y se le asigna ese valor en la posición que tenga i del arreglo.

Se propone al lector que realice la búsqueda del mínimo y máximo de la lista generada en este pequeño código.

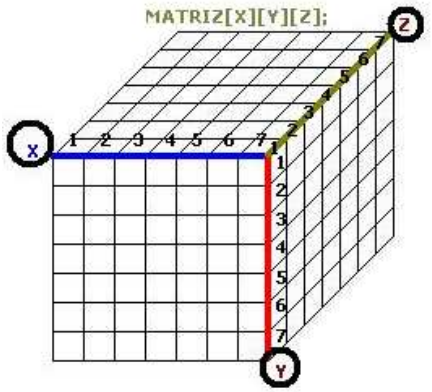
srand: Permite inicializar la serie de números pseudoaleatorios a partir de una "semilla" recibida como parámetro (en este caso se usa la hora del sistema)

A continuación se dará el concepto de arreglos multidimensionales, a partir del concepto de arreglo lineal puede extenderse agregando más dimensiones para dar lugar a los arreglos multidimensionales. La

implementación interna y la operatoria sobre este tipo de estructuras no cambia, salvo porque para acceder a una posición, el usuario debe proporcionar tantos subíndices como sea la dimensión del arreglo. Por ejemplo:



A la derecha se tiene una matriz de m x n lo cual se conoce como matriz bidimensional, en la parte izquierda se tiene otra matriz de tres dimensiones, lo cual se conoce como matriz tridimensional, la pregunta es ¿Cómo se crean dichas estructuras?



Se pondrá el código para inicializar la matriz bidimensional y tridimensional a 0.

```
int enteros[M][N];
int i, j;
for (i=0; i<M; i++)
 {
   for (j=0; i<N; j++)
     {
       enteros[i][j] = 0;
     }
 }
```

ón de la matriz bidimensional, se definen N y M
l tamaño de la matriz.

Creación de la matriz tridimensional.

rido de las filas.

Recorrido en el eje X.

rido de las columnas.

Recorrido en el eje Y.

osición i y j de la matriz
signa 0.

Recorrido en el eje Z.

```
M][N][O];

; i++)

<N; j++)

O;k<O;k++)

eros[i][j][k] = 0;
```

Punteros: *Un puntero es un tipo especial de variable, que almacena el valor de una dirección de memoria, esta dirección puede ser la de una variable individual, pero más frecuentemente será la de un elemento de un **array**, una estructura u objeto de una clase. Los punteros, al igual que una variable común, pertenecen a un tipo (**type**), se dice que un puntero 'apunta a' ese tipo al que pertenece. La declaración es el siguiente:*

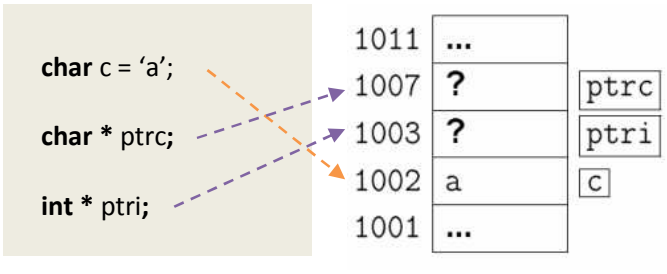
```
<tipo> * <nombre del puntero>;
```

Por ejemplo:

```
int * pint; //Declara un puntero a entero
char *pchar; //Puntero a un tipo char
```

Independientemente del tamaño (***sizeof***) del objeto apuntado, el valor almacenado por el puntero será el de una única dirección de memoria. En sentido estricto un puntero no puede almacenar la dirección de memoria de 'un ***array*** (completo), sino la de un elemento de un ***array***, y por este motivo no existen diferencias sintácticas entre punteros a elementos individuales y punteros a arrays. La declaración de un puntero a ***char*** y otro a ***array*** de ***char*** es igual.

Cuando se declara un puntero se reserva memoria para albergar una dirección de memoria, pero **NO** para almacenar el dato al que apunta el puntero. Ejemplo:



El operador "**&**" devuelve la dirección de memoria donde comienza la variable, también sirve para asignar valores a datos de tipo puntero como se observa a continuación.

```
int i = 10;
int *ptr;
.....
.....
ptr = &i;
.....
```

Indirección: *<ptr> devuelve el contenido del objeto referenciado por el puntero <ptr>. El operador * se usa para el acceso a los objetos a los que apunta un puntero. Por ejemplo:

```
char c ;
char * ptr;
....
ptr = &c;
*ptr = 'A';
```

Otro ejemplo puntero del mismo tipo.

Asignación de la dirección de la variable c.

```
char c ;
char * ptr1;
char * ptr2;
....
ptr1 = &c;
ptr2 = ptr1;
```

Creación de dos punteros del mismo tipo.

Asignación de la dirección de la variable c.

Asignación de punteros del mismo tipo.

A continuación se listan ciertos problemas resueltos para esta sección.

Ejercicios Resueltos

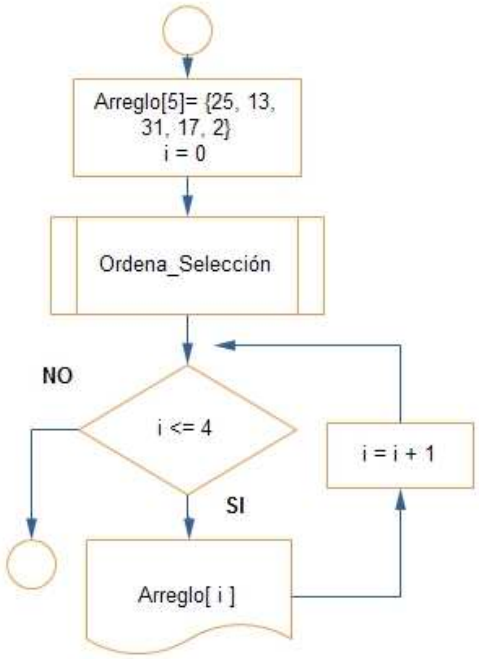
Problema 1: Escriba un programa en C, que realice la ordenación de un vector de una dimensión usando el método de "Ordenación de selección". Una ordenación de selección recorre un arreglo buscando el elemento más pequeño del mismo. Cuando encuentra el más pequeño, es intercambiado con el primer elemento del arreglo. El proceso a continuación se repite para el subarreglo que empieza con el segundo elemento del arreglo. Cada pasada del arreglo resulta en un elemento colocado en su posición correcta. Esta ordenación requiere de capacidades de procesamiento similares a la ordenación de tipo burbuja para un arreglo de n elementos, deberán de hacerse n-1 pasada, y para cada subarreglo, se harán n-1 comparaciones para encontrar el valor más pequeño. Cuando el subarreglo bajo procesa contenga un solo elemento, el arreglo habrá quedado terminado y ordenado.

Restricciones: El programa debe usar arreglos y funciones.

Algoritmo

Datos de Entrada:
Arreglo desordenado de 5 números enteros
Datos de Salida:
Arreglo ordenado de 5 números enteros
Algoritmo:
Inicio
Encuentra el valor más pequeño de la lista.
Intercambia el valor más pequeño encontrado con el valor de la primera posición del arreglo
Repita los pasos anteriores para los elementos restantes del arreglo (comience en la segunda posición y avance un elemento a la vez)
Fin

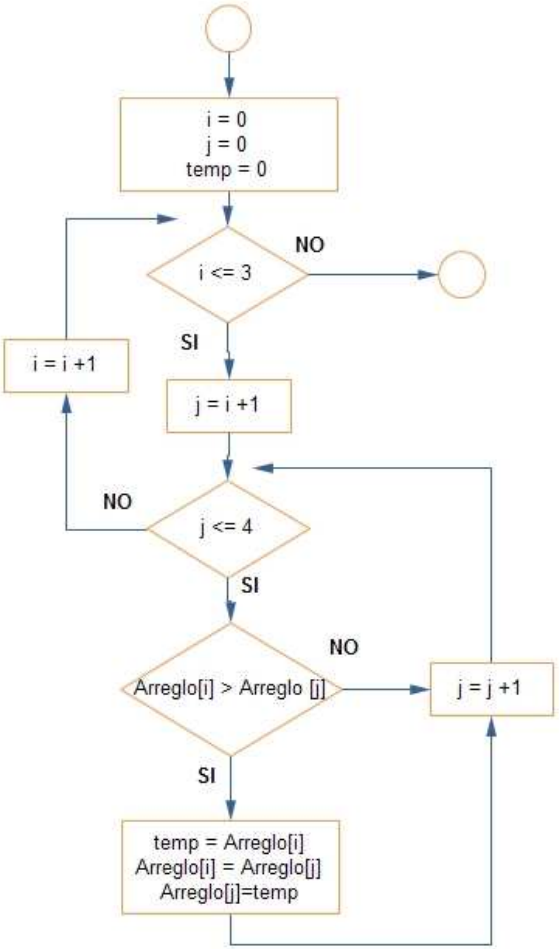
Diagrama de Flujo o Pseudocódigo



Código en C

```
#include <stdio.h>
#include <windows.h>

int Arreglo[5] = { 25, 13, 31, 17, 2 } ;

void Ordena_Seleccion();

void main( )
{
        int i = 0;
        printf ( "\n\n ORDENAMIENTO POR SELECCION \n") ;
   printf ( "\n\n Arreglo antes el ordenamiento:\n") ;
        for ( i = 0 ; i <= 4 ; i++ )
                printf ( "%d\t", Arreglo[i] ) ;
        Ordena_Seleccion();
        printf ("\n\nArreglo despues el ordenamiento:\n") ;
        for ( i = 0 ; i <= 4 ; i++ )
                printf ( "%d\t", Arreglo[i] ) ;
        Sleep(5000);
   return 0;
}


void Ordena_Seleccion()
{
    int i=0, j=0, temp=0;
```

A continuación se proporciona el procedimiento Ordena_Selección




```
        for ( i = 0 ; i <= 3 ; i++ )
        {
                for ( j = i + 1 ; j <= 4 ; j++ )
                {
                        if ( Arreglo[i] > Arreglo[j] )
                        {
                           temp = Arreglo[i] ;
                           Arreglo[i] = Arreglo[j] ;
                           Arreglo[j] = temp ;
                        }
                }
        }
}
```

Problema 2: Escriba un programa que encuentre el cuadrado mágico para cualquier n donde 3< n< 10 de orden impar.
Un cuadrado mágico es una cuadrícula en general de n x n, en la que se acomodan ciertos números que cumplen que la suma de cualquier renglón, la suma de cualquier columna y la suma de cualquiera de cualquiera de las dos diagonales es siempre la misma.
Si el cuadrado es de 3 x 3, entonces tendrá 9 casillas y los números que se acomodan en él son todos los números del 1 al 9.
Si el cuadrado es de 4 x 4, entonces tendrá 16 casillas y los números que se acomodan en él son del 1 al 16.
En general, si el cuadrado es de n x n, entonces tendrá n cuadrada casillas y los números que acomodaremos en él serán del 1 a n^2.
El orden de un cuadrado mágico es el número de renglones o el número de columnas que tiene. Así un cuadrado de 3 x 3 se dice que es de orden 3.

Restricciones: El programa debe usar arreglos y funciones.

Algoritmo

Datos de Entrada:

Orden del cuadrado mágico, el cual es un numero entero inpar

Datos de Salida:

Una matriz de números enteros de n x n

Algoritmo:

Inicio

Colocamos el número 1 en la celda central de la fila superior.

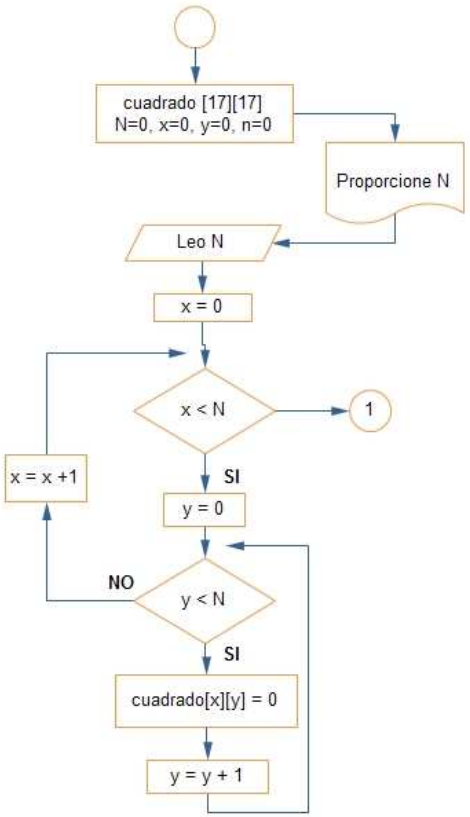
La cifra consecutiva a una cualquiera debe colocarse en la celda que le sigue diagonalmente hacia arriba y hacia la derecha.

Si al hacer esto se sale del cuadrado por el límite superior del contorno del mismo, saltaremos a la celda de la columna siguiente hacia la derecha y en su fila inferior, si se sale por la derecha, se sigue por la primera celda, a partir de la izquierda, de la fila superior.

Cuando la celda siguiente está ocupada, el número consecutivo de la serie se coloca en la celda inmediatamente inferior a la del número precedente, comenzando así un nuevo camino en la dirección de la diagonal.

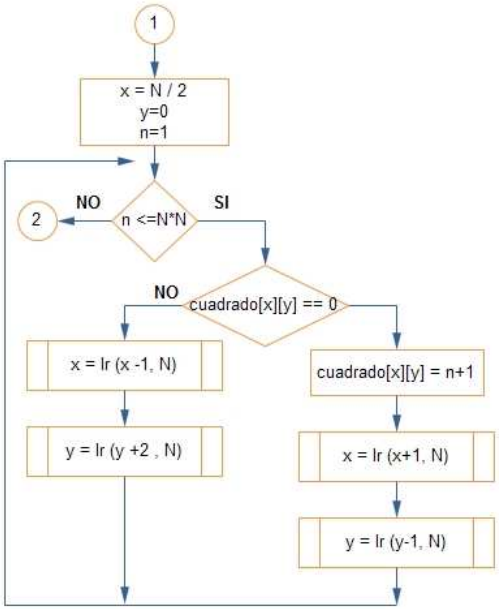
Fin.

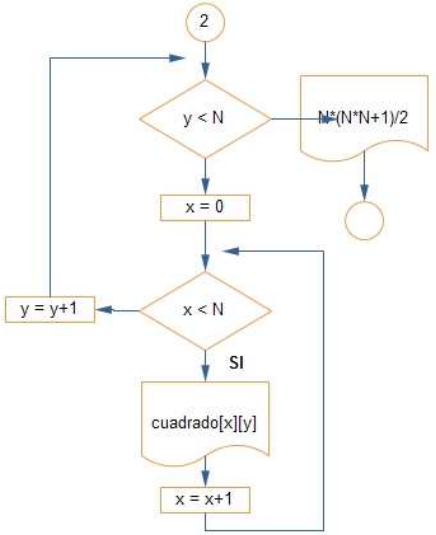
Diagrama de Flujo o Pseudocódigo



Código en C

```
#include <stdio.h>
#include <conio.h>

/*limitar rango ( limita los valores de x al rango [0, N] )*/
#define lr( x, N ) ( (x)<0 ? N+(x)%N : ( (x)>=N ? (x)%N : (x)
) )

int main()
{
 int cuadrado[17][17],x,y,n,N;

/*restricción del orden a los impares entre 3 y 17, por
motivos de dar mayor facilidad para entender el
código*/
 do{
  printf( "nIngrese el orden ( impar entre 3 y 17 ): " );
  scanf( "%i", &N );
 }while( !(N%2) );
 printf( "nCuadrado Mágico de orden %ix%i :nn", N, N);

/*Se inicia los elementos del cuadrado mágico con
ceros*/
 for(x=0;x<N;x++)
  for(y=0;y<N;y++)
   cuadrado[x][y]=0;

/*Se aplica el algoritmo general para obtener cuadrados
magicos de orden impar*/
 for(  x=N/2,y=0,n=1;  n<=N*N;  )  /*se hace  N*N
iteraciones...*/
```



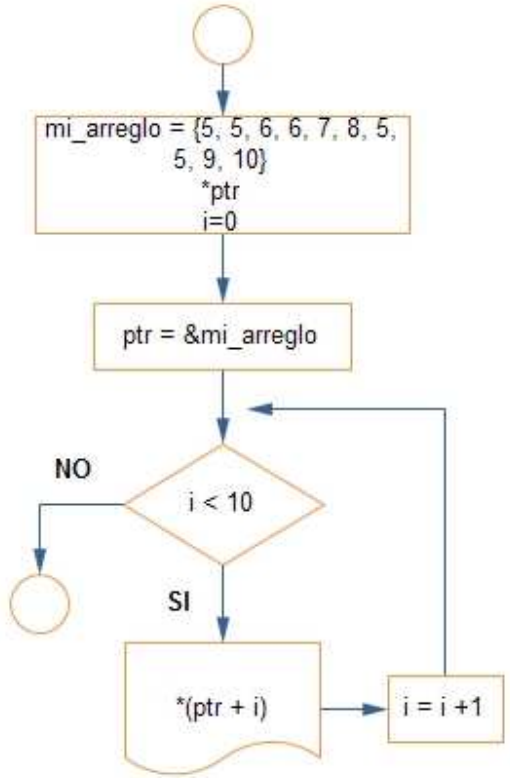



```
 if( !cuadrado[x][y] )  /*si el elemento seleccionado es cero*/
  cuadrado[x][y] = n++, /*se inserta un número natural */
  x=lr(x+1,N),         /*se incrementa x en 1          */
  y=lr(y-1,N);         /*se decrementa y en 1          */
 else x=lr(x-1,N),     /*se decrementa x en 1          */
      y=lr(y+2,N);     /*se incrementa y en 2          */

   /*se imprime el cuadrado magico en pantalla*/
for(y=0;y<N;y++)
{
 printf("n");
 for(x=0;x<N;x++)
  printf("%4i", cuadrado[x][y] );
}

printf("nn Suma = %inn", (N*(N*N+1))/2 );/*se imprime la suma*/

getch();
return 0;
}
```

Problema 3: A partir de una lista de calificaciones de los alumnos de primer grado, acceder y mostrar la información correspondiente a dicha lista mediante el acceso de un apuntador.

Restricciones: Uso de punteros, estructuras de control repetitiva y el arreglo inicia con diez calificaciones.

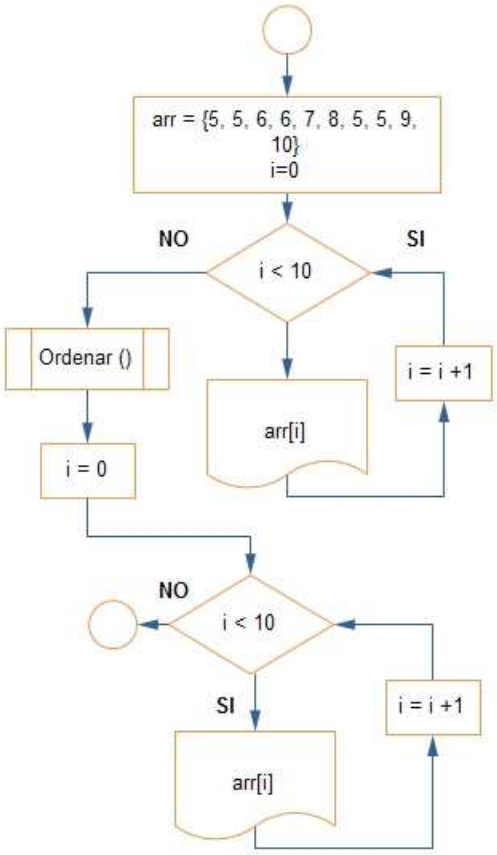
Algoritmo	Diagrama de Flujo o Pseudocódigo	Código en C

Datos de entrada:
 	
Datos de salida:
La información del arreglo mediante un puntero.
Algoritmo:
 	
inicio
Le asigno al mi arreglo creado inicialmente los valores correspondientes.
Posteriormente asigno al puntero la dirección de la posición de mi primer elemento del arreglo.
Mi índice lo inicializo en 0.
Repetir
Imprimir en pantalla el índice y el valor que está apuntando el puntero más el índice.
Incrementar mi índice.
Hasta que mi índice sea mayor a los 10 elementos de mi arreglo.
fin
		#include <stdio.h>
		//incializo mi arreglo con las calificaciones
		int mi_arreglo[] = {5, 5, 6, 6, 7, 8, 5, 5, 9, 9};
		//creo mi puntero
		int *ptr;
		
		int main ()
		{
		int i=0; //Mi indice
		ptr = &mi_arreglo[0]; //le asigno la dirección
		//del primer elemento
		//del arreglo
		for (i=0; i<10; i++)
		{
		printf("ptr + %d = %d \n", i, *(ptr +i));
		}
		}

Problema 4: Escribir una función que sea capaz de ordenar virtualmente una arreglo de enteros, las funciones deben contener algún manejo de puntero.

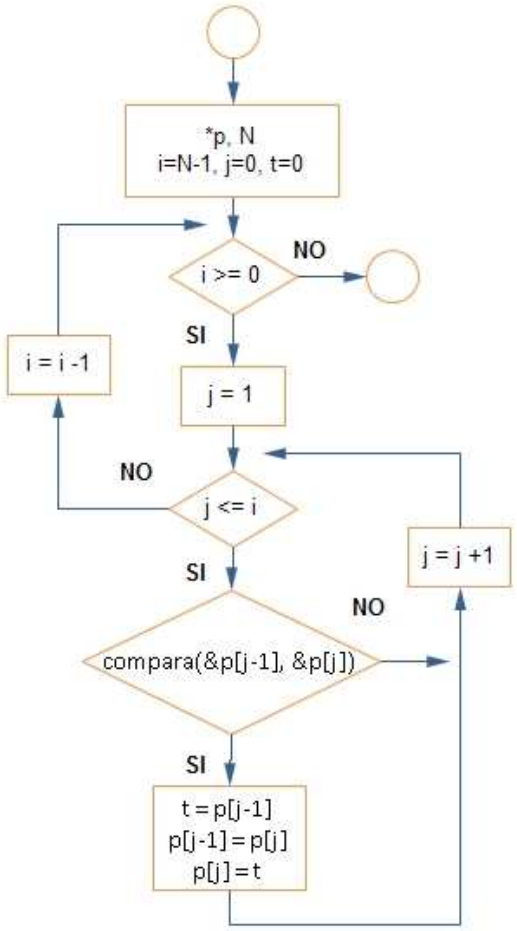
Restricciones: Uso de estructuras de control, funciones y apuntadores.

Algoritmo	Diagrama de Flujo o Pseudocódigo	Código en C

Datos de entrada:
 	
Datos de salida:
La información del arreglo ordenada mediante una función usando punteros.
Algoritmo:
 	
Inicio
Le asigno al mi arreglo creado inicialmente los valores correspondientes.
Imprimo todos los valores dados anteriormente.
Invoco el procedimiento ordena con el arreglo desordenado y el número de elementos.
Posteriormente imprimo todos los datos del arreglo para observar si están ordenados.
Fin
 	
Inicio
En mi procedimiento recibo como parámetro un puntero de tipo entero, el que será que apuntará a mi arreglo y la cantidad de elementos.
Para índice i igual N-1 hasta 0 hacer
Para índice j igual a 1 hasta i hacer
Si comparamos la dirección en el índice del arreglo j-1 y j
Realizo el intercambio que será
t le asigno p[j-1]
		#include <stdio.h>
		
		int arr[10]={2, 3, 4, 5, 9, 19, 300, 29, 4, 1};
		
		void ordena (int *p, int N);
		void compara (int *m, int *n);
		
		int main()
		{
		int i=0;
		for (i=0; i < 10; i ++)
		{
		printf("[%d] ", arr[i]);
		}
		
		ordena(arr, 10);
		printf("\n");
		
		for (i=0; i < 10; i ++)
		{
		printf("[%d] ", arr[i]);
		}
		getch();
		return 0;
		}
		
		void ordena (int *p, int N)
		{
		int i, j, t;

p[j-1] es igual p [j]
p[j] es igual a t
Incrementar índice j
Decremento i
Fin de índice j
Fin de índice i.
fin

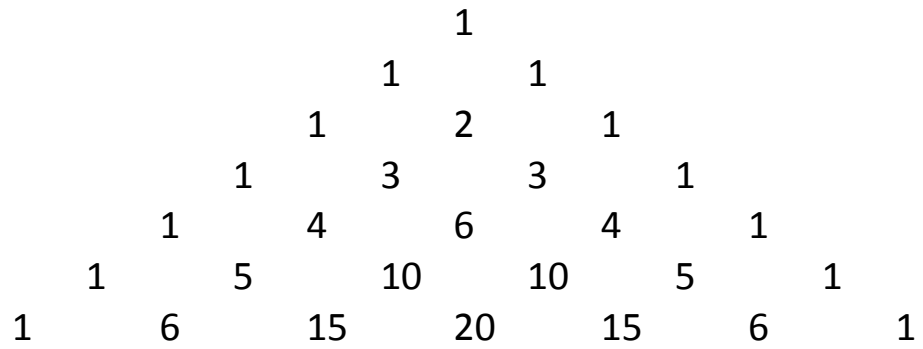
Función Ordena




```
  for (i=N-1; i >=0; i--)
    {
      for (j = 1; j<=i; j++)
      {
        if (compara(&p[j-1], &p[j]))
        {
          t = p[j-1];
          p[j-1]= p[j];
          p[j] = t;
        }
      }
    }
}

int compara(int *m, int *n)
{
  return (*m >*n);
}
```

Ejercicios Propuestos

Problema 1: Escribir un programa que permita visualizar el triangulo de pascal. En el triángulo de pascal cada número es la suma de los dos números situados encima de él. Este problema se debe resolver utilizando un arreglo de una sola dimensión.



Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

Problema 2: El juego del ahorcado se juega con dos personas (o una persona y una computadora). Un jugador selecciona una palabra y el otro jugador trata de adivinar la palabra adivinando las letras individuales. Diseñar un programa para jugar al ahorcado.

Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

Problema 3: Se dice que una matriz tiene un punto de silla si alguna posición de la matriz es el menor valor de su fila, y a la vez el mayor de su columna. Escribir un programa que tenga como entrada una matriz de números reales, y calcular la posición de un punto de silla (si es que existe).

Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

Problema 4: Se desea escribir un programa que permita manejar la información de habitantes de un complejo habitacional. El mismo posee 7 torres; a su vez cada torre posee 20 pisos y cada piso 6 departamentos.
Se desea saber:

a- Cantidad total de habitantes del complejo

b- Cantidad promedio de habitantes por piso de cada torre
c- Cantidad promedio de habitantes por torre

Restricciones: Uso arreglos, funciones, así como las estructuras de control selectivas y repetitivas.

Problema 5: Implemente la multiplicación de matrices utilizando punteros.

Restricciones: Uso de punteros, funciones, así como las estructuras de control selectivas y repetitivas.

Problema 6: Sea A una matriz de tamaño nxn, implemente un programa que dado un menú de opciones resuelva:

- La transpuesta de A (At).
- Si A es simétrica o antisimétrica.
- Si A es una matriz triangular superior o triangular inferior.

Restricciones: Uso de punteros, funciones, así como las estructuras de control selectivas y repetitivas.