

Capítulo 1.

Introducción a las Estructuras de Datos y Algoritmos.

1.1. Estructura de los datos.

La información se representa mediante bits. Los bits se organizan en bytes y palabras de memoria, mediante las cuales se representan los **tipos básicos de datos**.

Mediante los mecanismos de **agrupación** en estructuras y arreglos se pueden seguir elaborando estructuras más complejas; mediante la **vinculación**, empleando cursores o punteros, se pueden establecer relaciones entre componentes de la estructura de datos.

Con estos elementos pueden describirse **estructuras abstractas de datos** como: listas, árboles, conjuntos, grafos. Usando estructuras abstractas de datos se pueden modelar los datos de sistemas más complejos, como: sistemas operativos, la información que viaja en los paquetes de datos de una red, o complejas relaciones entre componentes de datos en un sistema de bases de datos, por mencionar algunos ejemplos.

Las relaciones de los elementos de datos, en sus diferentes niveles, pueden conceptualizarse en el siguiente diagrama:

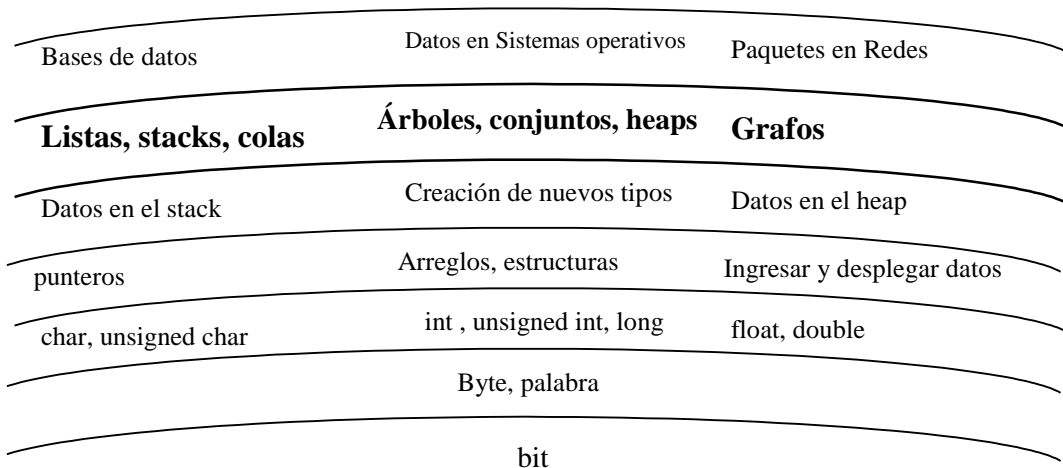


Figura 1.1. Estructuras de datos.

Los dos niveles inferiores son cubiertos en asignaturas sobre sistemas digitales.

Los dos niveles siguientes deberían cubrirse en un curso básico de programación.

Los cuatro niveles inferiores también son tratados en cursos de estructuras de computadores, en el cual se elaboran las estructuras de datos del lenguaje de alto nivel, mediante los mecanismos de estructuración de datos y direccionamientos assembler; los que a su vez son explicados en términos de los modos de direccionamiento de las instrucciones de máquina; en este nivel también se describe el uso de los diferentes registros y segmentos de memoria, y la configuración detallada de los frames en el stack.

Los niveles quinto y sexto son el tema de este texto.

1.2. Estructura de las acciones. Algoritmos.

En su forma más primitiva la agrupación de compuertas permite el desarrollo de acciones combinatorias, entre ellas: la unidad aritmética y lógica, unidades de corrimiento, extensores, muxes. Con la ayuda de elementos de memoria y conducidas por un reloj, se pueden elaborar máquinas secuenciales que efectúen las acciones de contar, desplazar, multiplicar y dividir, así también el generar las acciones de control que gobiernan las transferencias entre los recursos básicos de un procesador. Esto se cubre en un curso de Sistemas Digitales.

Luego de esta estructuración, las acciones realizadas electrónicamente pueden ser abstraídas como instrucciones de máquina. Mediante la ayuda de compiladores es posible traducir las acciones o sentencias del lenguaje de alto nivel en instrucciones de máquina. En el nivel de los lenguajes de programación, se dispone de operadores que permiten construir expresiones y condiciones; agregando las acciones que implementan las funciones combinatorias se pueden abstraer las acciones en términos de alternativas, condicionales y switches; la capacidad de implementar máquinas de estados permite la realización de iteraciones y repeticiones. Las organizaciones y arquitecturas se estudian en un curso de estructuras de computadores.

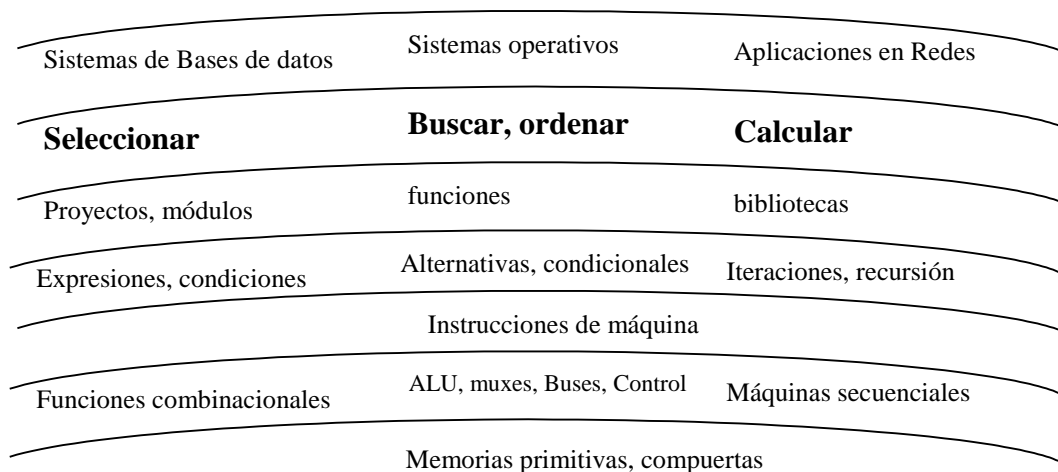


Figura 1.2. Estructura de acciones.

El siguiente nivel es el de un curso de programación, el que debería cubrir: las reglas para la construcción de expresiones, condiciones, alternativas y repeticiones; también la correcta forma de diseñar funciones pasando los parámetros con los tipos adecuados, y la elección de las variables locales y globales; así también la forma de comunicar los resultados de las funciones. En un curso básico de programación también se domina el empleo de las bibliotecas de entrada-salida y las matemáticas.

El tema cubierto en este texto centra su atención en el diseño de algoritmos para realizar las acciones básicas de: **Ordenar, buscar, seleccionar y calcular**, empleando diferentes estructuras de datos. Por ejemplo, se pueden ordenar los elementos de una lista o de un arreglo; se puede buscar un valor en un árbol, en una lista o en un arreglo.

La elección adecuada de la estructura de datos y del algoritmo empleado permite obtener un diseño eficiente, tanto en recursos ocupados como en el tiempo de ejecución. Parte importante del curso estará centrada en lograr una medida de la eficiencia de cada algoritmo y su comportamiento a medida que aumenta el número de elementos que constituyen los datos; es decir, de su **complejidad**.

Estos algoritmos eficientes tienen importantes aplicaciones: en el diseño de sistemas operativos, en la programación de sistemas y de aplicaciones en red, en la elaboración de sistemas de bases de datos, en la construcción de compiladores y aplicaciones multimediales, por nombrar las más importantes.

1.3. Lenguajes.

En los diferentes niveles de datos y acciones se emplean diferentes lenguajes.

En los niveles físicos, se emplean Lenguajes de Descripción de Hardware (HDL). Para describir los procesadores se emplea lenguaje assembler, cada procesador tiene un lenguaje assembler propio.

En lenguajes de alto nivel ha existido una rápida evolución. Muchos de los que fueron más universalmente usados han pasado a la historia: Fortran, Algol, APL, Pascal, Modula, Basic.

Los primeros lenguajes ofrecían al programador la posibilidad de efectuar saltos. El primer avance evolutivo fundamentado fue impedir que el programador pudiese emplear los saltos, y que aceptase formas estructuradas de organizar las acciones. Es decir que construyera algoritmos organizando las acciones en secuencias, alternativas o iteraciones.

El siguiente modelo conceptual fue el desarrollo modular en el cual se definían estructuras de datos y las funciones que las manipulaban; es decir la concepción de tipos abstractos de datos.

Los lenguajes como Pascal, C y otros tienen la posibilidad de programación de tipos abstractos. El lenguaje C evolucionó rápidamente para convertirse en uno de los más usados. Su principio de diseño es que compile eficientemente y que permita usar los recursos de hardware de los procesadores. Por esta razón se lo seguirá empleando en cursos de estructuras de computadores y en el diseño de microcontroladores. Sin embargo sus capacidades de manejo de memoria

dinámica y el empleo de punteros, que son sus principales ventajas, son sus principales fuentes de errores en la elaboración de grandes programas.

El modelo actual es el diseño y la programación orientada a objetos, en el cual se impide al programador emplear punteros, y el manejo de la memoria dinámica es responsabilidad del lenguaje y no del programador. Los datos y las acciones conforman el objeto, no están separadas. Existen varios de estos lenguajes, Java es uno de los que más se ha difundido.

Seguramente con los años aparecerán nuevos y mejores lenguajes, pero los principales conceptos que aprendamos en este texto seguramente perseverarán, ya que son básicos.

1.4. Genialidades.

Los más importantes algoritmos que estudiaremos emplean ideas geniales y brillantes. Desde que son descubiertos se los empieza a usar en las aplicaciones, debido a sus ventajas. Sin embargo exponer ideas: a veces sofisticadas, otras decididamente rebuscadas, en algunos casos aparentemente simplistas, y que han resuelto grandes problemas son parte de la dificultad de este curso.

No se estudian cuestiones sencillas, la mayor parte de ellas son muy elaboradas, y algunas veces difíciles de captar en su profundidad.

Debido a lo anterior, no se espera que después de estudiar este texto se esté en condiciones de inventar algoritmos que pasen a la historia, pero si conocer las principales estructuras abstractas de uso general, sus posibles aplicaciones, y los mejores algoritmos conocidos para esas estructuras.

1.5. Teoría y práctica.

La formación de un ingeniero debe contemplar un balance adecuado entre la teoría y la práctica. Por un lado se requiere disponer de un marco conceptual que permita pensar y reflexionar sobre un determinado problema; por otro, la capacidad de aplicar las ideas a situaciones reales.

En el caso particular de este curso, la capacidad de realización la entenderemos como el conjunto de diseñar estructuras de datos y algoritmos eficientes, mediante el lenguaje de programación C; la depuración de programas, la verificación de su funcionamiento correcto, a través de someterlos a datos de prueba escogidos convenientemente.

1.6. Definiciones.

1.6.1. Algoritmo.

Secuencia finita de operaciones, organizadas para realizar una tarea determinada.

Cada operación debe tener un significado preciso y debe ser realizada en un lapso finito de tiempo y con un esfuerzo finito. Un algoritmo debe terminar después de ejecutar un número finito de instrucciones.

Algoritmos diferentes pueden completar la misma tarea con la ejecución de un conjunto de instrucciones diferentes, en más o menos tiempo, y empleando más o menos memoria. Es decir, pueden tener complejidad y costo diferentes.

Se pueden analizar algoritmos, en forma abstracta, es decir sin emplear un determinado lenguaje de programación; el análisis se centra en los principios fundamentales del algoritmo y no en una implementación en particular. En estos casos puede emplearse para su descripción un **pseudocódigo**.

Existen métodos cuantitativos para determinar los recursos espaciales y temporales que requiere un algoritmo. En particular se estudiarán métodos para calcular la **complejidad temporal** de un algoritmo.

1.6.2. Heurística.

Un algoritmo que produce soluciones razonablemente rápidas y buenas, pero no necesariamente la solución óptima.

1.6.3. Estructuras de datos.

La forma en que se organizan los datos para ser usados.

Es una colección de variables, posiblemente de diferentes tipos de datos, conectadas de un modo determinado.

Una estructura de datos bien organizada debe permitir realizar un conjunto de acciones sobre los datos de tal forma de minimizar el uso de los recursos y el tiempo empleado para efectuar la operación.

1.7. Algoritmos clásicos.

Se describen algunos algoritmos, considerados clásicos, para ilustrar el camino que debe recorrerse desde su diseño hasta traducirlo a un lenguaje de programación.

En opinión del Profesor Dijkstra, las computadoras manipulan símbolos y producen resultados; y un programa, que describe un algoritmo, es un manipulador abstracto de símbolos.

Visto de este modo un algoritmo es una fórmula de algún sistema formal; sin embargo debido a que estas fórmulas resultan mucho más largas y elaboradas que las usuales, no suele reconocérselas como tales.

En primer lugar debe describirse lo que se desea realizar en lenguaje natural, desgraciadamente esto puede producir más de alguna ambigüedad. Luego debe modelarse matemáticamente la situación que interesa describir; es decir, apoyado en conceptos matemáticos y lógicos se describe la función o fórmula que debe realizarse, empleando un pseudolenguaje; que es más preciso y formal que el lenguaje común. Una vez obtenida una representación formal se la traduce empleando un lenguaje de programación.

No es sencillo derivar el algoritmo desde un programa que lo implementa. Lo cual puede observarse intentando leer programas, sin disponer antes de la descripción del algoritmo.

Sin embargo el paso de la descripción en pseudocódigo a un lenguaje de programación suele ser una actividad de menor complejidad; además, esto permite su codificación en diferentes lenguajes de programación.

1.7.1. Algoritmo de Euclides.

Euclides fue un matemático griego que vivió alrededor del año 300 a.C. una de sus contribuciones es el algoritmo para *obtener el máximo común divisor de dos números enteros*, lo cual sería la descripción en lenguaje natural, del problema que se desea resolver.

Luego intentamos modelar matemáticamente la situación.

Si x e y son enteros, no ambos ceros, su máximo común divisor, que anotaremos $\text{mcd}(x,y)$, es el mayor entero que los divide a ambos exactamente; es decir, sin resto.

Ejemplo: $\text{mcd}(7, 11) = 1$
 $\text{mcd}(16, 28) = 4$

Puede comprobarse que:

$$\begin{aligned}\text{mcd}(x, 0) &= |x| \\ \text{mcd}(x, y) &= \text{mcd}(y, x) \\ \text{mcd}(-x, y) &= \text{mcd}(x, y)\end{aligned}$$

De lo cual se desprende que es de interés obtener un algoritmo para el mcd de enteros no negativos, el conocimiento matemático que se necesita, para resolver este problema es el siguiente:

Si los números son x e y :

- a) Si x es igual a y ; entonces x (ó y) es el resultado.
- b) Si se reemplaza el número mayor por la diferencia del mayor menos el menor, no cambia el máximo común divisor.

Lo anterior puede plantearse en términos matemáticos, según:

- a) $\text{mcd}(x, x) = x$
- b) Si $x > y$ se tiene $\text{mcd}(x, y) = \text{mcd}(x-y, y)$

Entonces para enteros mayores que cero, tenemos el siguiente algoritmo, descrito en lenguaje natural:

Mientras los números sean diferentes:

Deje el menor, y forme otro restando el menor al mayor.

Lo cual puede plantearse, empleando el lenguaje C:

```
while (x!=y) if (x>y) x=x-y; else y=y-x;
```

Como veremos, a través de los siguientes ejemplos, existen numerosas formas de describir un algoritmo.

Otra forma de describir el Algoritmo de Euclides es la siguiente:

```
do
{
  while (x>y) x=x-y;
  while (y>x) y=y-x;
} while (x!=y);
// x es el mcd
```

Veremos otra forma del algoritmo.

Debido a que puede comprobarse que:

$$\text{mcd}(x, y) = \text{mcd}(y, x \bmod y)$$

La expresión muestra cómo reducir uno de los números, manteniendo en el primer lugar al mayor. La transformación debe repetirse hasta que el menor de los números sea igual a cero. El cambio de posición de los números requiere una variable temporal que denominaremos resto.

Entonces una variante del algoritmo de Euclides es la siguiente:

```
while (y!=0)
{ resto = x % y;
  x=y;
  y=resto;
}
// x es el mcd
```

La siguiente función iterativa resume las ideas anteriores:

```
int mcd(int x, int y)
{ int resto;
  if (x==0) return (y);
  if (y==0) return (x);
  while (y!=0)
  { resto=x-y*(x/y); //resto=x % y;
    x=y;
    y=resto;
  }
  return (x);
}
```

Debido a que puede definirse en forma recursiva la función máximo común divisor, según:

$$\begin{aligned} \text{mcd}(x, 0) &= |x|, \\ \text{mcd}(x, y) &= \text{mcd}(y, x \bmod y) \end{aligned}$$

Se puede describir el algoritmo de Euclides, en forma recursiva:

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

1.7.2. Algoritmo el colador de Eratóstenes.

Eratóstenes fue un matemático griego (275-194 A.C.) que desarrolló un algoritmo que se conoce como la criba de Eratóstenes (sieve en inglés), que permite *determinar los números primos menores que un número dado*. El nombre deriva de un “colador” en que se colocan todos los números, y sólo deja pasar los números primos.

Para desarrollar el modelo matemático que permita elaborar el algoritmo, pueden emplearse las siguientes definiciones de un número primo:

Un número primo es un entero positivo que no es el producto de dos enteros positivos menores.

Un número primo es un entero positivo que tiene exactamente dos factores enteros positivos: 1 y sí mismo.

Entonces el algoritmo, puede describirse informalmente, según:

Se crea arreglo de números booleanos con el tamaño de los números que se desea probar si son o no primos y se los marca como primos. El índice del arreglo está asociado al número entero.

Se marca el elemento con índice 1 como no primo. Ya que no existen dos enteros diferentes menores que 1.

Se recorre el arreglo, en forma ascendente, dejando el siguiente número, previamente marcado como primo al inicio, como primo; y todos los múltiplos de éste, se marcan como no primos.

Puede comprobarse que el primer múltiplo a eliminar es el cuadrado del número.

El siguiente programa ilustra el algoritmo, se ha agregado la función mostrar para efectuar un listado de los números primos. Se listan los primos menores que la constante n .

```
#include <stdio.h>
#define n 40
#define primo 1
#define noprimo 0
```

```
int a[n];
```

```
void mostrar(void)
{ int i;
  for (i= 2; i<=n; i++)
    if (a[i]==primo) printf(" %d ", i);
}

void Eratostenes(void)
{ int i, p, j;
  a[1] = noprimo; for (i= 2; i<=n; i++) a[i]= primo;
//Se inicia arreglo, marcando todos los números como primos, excepto el primero.
  p = 2;
  while (p*p <= n)
  {
    j = p*p; //El primer múltiplo de p no primo es su cuadrado.
    while (j <= n)
    {
      a[j] = noprimo;
      j = j+p; //siguiente múltiplo de p
    }
    //printf("Elimina múltiplos de %d: ", p); mostrar(); putchar('\n');
    do p = p+1; while (a[p] == noprimo); //se salta los no primos.
  }
}

int main(void)
{
  Eratostenes();
  mostrar();
  return (0);
}
```

La criba de Atkin es un algoritmo moderno, más rápido, para encontrar todos los primos hasta un entero dado

1.7.3. Permutaciones en orden lexicográfico. Algoritmo de Dijkstra (1930-2002).

La permutación f precede a la permutación g en orden lexicográfico si y sólo si para el mínimo valor de k tal que $f(k) \neq g(k)$, se tiene $f(k) < g(k)$.

Si suponemos un arreglo de dígitos enteros, con índices desde 1 a n , donde en n se almacena el dígito menos significativo de la secuencia, y en la posición 1, el dígito más significativo. La generación de la permutación siguiente a la del arreglo inicial, está basada en encontrar el dígito que sea menor que el siguiente, a partir del dígito menos significativo; luego se busca a partir del último el primer dígito que sea mayor que el menor ya seleccionado, y se los intercambia. Luego se reordenan, en orden creciente, los dígitos restantes a partir del siguiente a la posición donde se depositó el mayor.

Para entender el algoritmo suponemos un arreglo de enteros en el cual almacenamos los dígitos: 1, 3, 2, 5 y 4 y deseamos generar la permutación siguiente a la almacenada.

Si tenemos la permutación: 13254, el dígito dos es menor que 5, y el primero mayor que 2 es 4. Resulta, luego del intercambio: 13452. Lo que resta es reordenar los dígitos siguientes a 4, hasta la posición del último, es este ejemplo hay que intercambiar el 2 con el 5, resultando: 13425 que es la permutación siguiente, en orden lexicográfico, a la inicial.

Veamos otro caso, a partir de: 15342. El primero menor que el dígito siguiente es el 3; y el primero mayor que 3 es el dígito 4; luego del intercambio, se tiene: 15432. Y reordenando los dígitos siguientes a 4, se obtiene la siguiente permutación: 15423.

Encontrar el dígito menor que el siguiente, a partir de la posición menos significativa, puede escribirse:

```
i = n-1; while (a[i] >= a[i+1]) i--;
```

Al salir del while se cumple $a[i] < a[i+1]$

Donde $a[i]$ es el menor seleccionado.

Nótese que si i es menor que uno, al salir del while, la secuencia es monótonamente decreciente, y no existe una permutación siguiente. Se asume que en la posición 0 del arreglo se ha almacenado un dígito menor que los dígitos almacenados en las posiciones de 1 a n . Esto es importante ya que debe asegurarse que el while termine, y también porque el lenguaje C no verifica que el índice de un arreglo esté dentro del rango en que ha sido definido.

Si existe la próxima permutación: Se busca a partir del último, el primer dígito que sea mayor que el menor ya seleccionado, lo cual puede traducirse según:

```
j = n; while (a[i] >= a[j]) j--;
```

Al salir del while se cumple $a[i] < a[j]$

Para el intercambio de los dígitos en posiciones i y j , se requiere una variable auxiliar *temp*, el uso del siguiente macro simplifica la escritura del intercambio.

```
#define swap(i, j) temp=a[i], a[i] = a[j], a[j] = temp
```

El reordenamiento en orden creciente, está basado en la observación de que la secuencia de números a reordenar es monótonamente decreciente, y por lo tanto basta intercambiar el primero con el último de la subsecuencia y así sucesivamente. No importando si el número de componentes es par o impar. El siguiente segmento reordena en forma ascendente, la secuencia descendente, desde $i+1$ hasta n ;

```
j = n;
i = i+1;
while (i < j)
{ swap(i, j);
  j--; i++;
```

```
}

```

El reordenamiento ascendente de la subsecuencia descendente, puede codificarse, en forma más compacta, empleando una sentencia for, en la que se emplea el operador secuencial coma:

```
for( i++, j = n; i < j ; i++, j--) swap(i, j);
```

La función getNext obtiene la siguiente permutación, de n dígitos, a partir de una dada, almacenada en el arreglo a . El resultado queda en el arreglo a . Debe prevenirse invocarla cuando el arreglo almacena la permutación mayor.

Si se decide que el retorno de la función sea verdadero si se generó la próxima permutación y un retorno falso implica que no se puede generar la siguiente, porque es la última, se tiene:

```
int getNext(int a[], int n)
{ int i, j, temp;
  a[0] = -1; //centinela
  for( i = n-1; a[i] >= a[i+1]; i--); if( i < 1) return ( 0);
  for( j = n; a[j] <= a[i]; j--);
  swap(i, j); // intercambia valores en posiciones (i) y (j)
  for( i++, j = n; i < j ; i++, j--) swap(i, j); return(1); //reordena
}
```

La siguiente función calcula, en forma iterativa, el número de permutaciones.

//Calcula n factorial. Número de permutaciones

```
long int P(int n)
{ int j; long int per=1L;
  for( j=1; j <= n; j++) per=per*j;
  return(per);
}
```

Se ha usado un entero largo para el resultado, debido al rápido crecimiento de la función factorial.

Problemas resueltos.

PI. Combinaciones en orden lexicográfico.

Una combinación es una colección no ordenada de elementos únicos. Si se tiene un conjunto C de elementos únicos (no repetidos) una combinación es un subconjunto de C .

Si tenemos que el conjunto C está formado por los dígitos $\{1, 2, 3, 4, 5\}$ las combinaciones formadas con tres elementos del conjunto, en orden creciente numérico, son la 6 siguientes:

```
123
124
125
134
```

135

145

Se inspecciona la cifra más significativa de la combinación, y si es menor que el mayor número del conjunto C, se incrementa la última cifra en uno, generando la combinación siguiente. Pero si la última es igual al último del conjunto, debe inspeccionarse la cifra anterior.

Se inspecciona la cifra anterior de la combinación, hasta encontrar una cifra menor a la máxima en esa posición. Una vez encontrada, ésta se incrementa en uno, y se completa la combinación con los sucesores.

Podemos describir con más detalle: si la combinación tiene k elementos y el conjunto tiene n elementos. En la posición k -ésima de la combinación el mayor número que puede escribirse es el n -ésimo del conjunto. En la posición $(k-1)$ -ésima de la combinación el mayor número que puede escribirse es el $(n-1)$ -ésimo del conjunto. La última combinación tiene en la posición 1 de la combinación el número $(n-k+1)$ del conjunto.

Con $n=7$ y $k=5$, si se tiene la combinación 12567, la segunda cifra de la combinación es menor que el máximo en esa posición, que corresponde a 4. Entonces se incrementa el 2 en uno, y se completa la combinación, generando: 13456. En este caso, la última combinación que puede generarse es 34567.

El número de combinaciones de k elementos sobre un total de n , está dado por $C(n, k)$ donde C es el coeficiente binomial.

Se tiene la siguiente definición recursiva del coeficiente binomial.

$$\begin{aligned} C(n, 0) &= 1 & n >= 0 \\ C(n, k) &= 0 & n < k \\ C(n, k) &= C(n-1, k) + C(n-1, k-1) & 0 <= k <= n \end{aligned}$$

La siguiente función efectúa el cálculo en forma recursiva:

```
int Cr(int n, int k)
{
  if (k==0) return (1); //asume n>=0
  else
    if (n<k) return (0);
    else return(Cr(n-1, k) + Cr(n-1, k-1));
}
```

La siguiente definición del coeficiente binomial puede implementarse en forma iterativa.

$$C(n, k) = n! / k!(n-k)!$$

Que puede escribirse:

$$C(n, k) = (n-k+1)(n-k+2)..(n) / 1*2*3*..*k$$

La cual puede codificarse según:

```
int C(int n, int k)
{ int j; long int num, den;
  num=1L; den=1L;
  for (j=1; j<=k; j++)
    { num=num*(n+1-j);
      den=den*j;
      //printf("%ld %ld\n", num,den);
    }
  return(num/den);
}
```

Se emplean enteros largos para *num* y *den* que crecen rápidamente.

La siguiente función, muestra los elementos del arreglo, desde 1 a *n*.

```
void mostrarcomb(int a[], int n)
{ int i;
  for (i=1; i<=n; i++) printf( " %2d", a[i] );
  putchar('\n');
}
```

Si suponemos un arreglo de enteros únicos (no repetidos), con índices desde 1 a *n*, y valores entre 1 y *n*, la generación de las combinaciones de *k* elementos, mayores en orden lexicográfico que la inicial, pueden generarse mediante:

```
void combine(int a[], int n, int k)
{ int j,i=k,t;
  mostrarcomb(a,k);
  while(1)
  {
    t=n-k; while (a[i]==t+i && i>0 ) i--;
    //La posición i marca una cifra menor a la máxima en esa posición.

    if(i==0) break; //para asegurar el término del lazo while externo
    t=a[i]-i+1; // completa la combinación con los sucesores.
    for(j=i; j<=k; j++) a[j]=t+j;

    mostrarcomb(a,k);
    i=k; /revisa la siguiente.
  }
}
```

La función que permite generar la siguiente combinación, a partir de una inicial, puede extraerse del algoritmo anterior.

```
void getNextC(int a[], int n, int k)
{ int j, i=k, t=n-k;
  while (a[i]==t+i && i>0) i--;
  t=a[i]-i+1; for(j=i; j<=k; j++) a[j]=t+j;
}
```

Debe evitar invocarse a la función pasando como dato inicial, la última combinación.

El siguiente segmento, genera todas las combinaciones, invocando a getNextC.

```
for (i=0; i <= N; i++) a[i] = i;
  mostrarcomb(a, k); //Muestra combinación inicial
  j=Cr(N, k);
  for (i=0; i < j-1; i++)
    {getNextC(a, N, k); mostrarcomb(a, k);}
```

La siguiente fórmula genera todas las combinaciones de k elementos de un conjunto de n elementos únicos. Obtener el algoritmo empleado, a partir del código no resulta una tarea sencilla, como puede verificarse.

```
void combine(int a[], int n, int k)
{
  int p, b;
  mostrarcomb(a, k); //escribe la inicial
  p=1;
  while (p<=k)
  {
    if (a[k+1-p] < n+1-p) //si es menor genera la siguiente
    {
      b=a[k+1-p]; //en b recuerda la cifra que debe incrementarse en uno
      while (p>=1)
      {
        a[k+1-p]=b+1; //produce secuencia de sucesores.
        b++;
        p--;
      }
      mostrarcomb(a, k);
      p=1;
    }
    else p++; //si es igual aumenta p
  }
}
```

P2. Cálculo del recíproco.

En los primeros computadores no estaba implementada la operación de división, en algunos sólo se disponía de sumas y multiplicaciones. Se diseñaron algoritmos para implementar las funciones para obtener el recíproco y la raíz cuadrada de un número real mediante sumas y multiplicaciones.

Desarrollaremos el algoritmo para obtener el recíproco, de un número real positivo a , como la raíz de $f(x)$:

$$f(x) = \frac{1}{x} - a$$

Emplearemos la aproximación de Newton, que genera el valor $(k+1)$ -ésimo mediante el punto de intersección de la tangente a la función, en el valor k -ésimo de x , con el eje x .

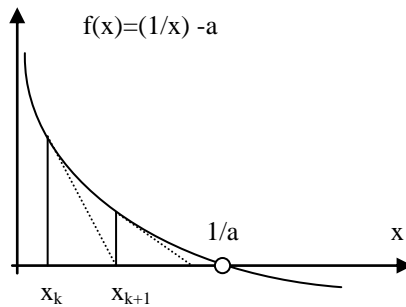


Figura 1.3. Secuencia de aproximación usando el método de la tangente.

Debe notarse que si el valor inicial de x se escoge muy grande, no se producirá el acercamiento a la raíz. La pendiente en $x = 2/a$, intercepta al eje x en cero, y en este punto la función tiene derivada con valor infinito; por lo tanto puede escogerse un valor inicial en el intervalo: $0 < x_0 < 2/a$, para garantizar la convergencia.

De la Figura 1.3, se obtiene para la tangente:

$$f'(x_k) = \frac{f(x_{k+1}) - f(x_k)}{x_{k+1} - x_k} \approx -\frac{f(x_k)}{x_{k+1} - x_k}$$

Que permite obtener, la recurrencia:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Reemplazando la función y su derivada, evaluadas en el punto actual, se obtiene:

$$x_{k+1} = x_k - \frac{\frac{1}{x_k} - a}{-\frac{1}{x_k^2}} = 2x_k - ax_k^2$$

Debe notarse que el nuevo punto de aproximación se obtiene empleando solamente sumas y multiplicaciones.

Si definimos el error relativo de la aproximación, en la iteración k -ésima, mediante:

$$e_k = \frac{\left| x_k - \frac{1}{a} \right|}{\frac{1}{a}} = |1 - ax_k|$$

Reemplazando la relación de recurrencia, se obtiene:

$$e_k = |1 - a(2x_{k-1} - ax_{k-1}^2)| = (1 - ax_{k-1})^2 = e_{k-1}^2$$

Relación que muestra que el error disminuye cuadráticamente. Si el error es una etapa es pequeño, en la etapa siguiente disminuye mucho más.

Expresando en términos del error relativo, se obtienen:

$$x_k = (1 + 1 - ax_{k-1})x_{k-1} = (1 + e_{k-1})x_{k-1}$$

$$e_k = e_{k-1}^2$$

Si el valor inicial se escoge, dentro del intervalo de convergencia, por ejemplo en $x = 1$, se tienen los valores iniciales:

$$x_0 = 1$$

$$e_0 = 1 - a$$

La codificación de un algoritmo, basado en relaciones de recurrencia, resulta sencilla observando que las variables que figuran a la izquierda de las asignaciones corresponden a las con subíndice $(k-1)$ y las a la derecha de las asignaciones corresponden a las variables con índice k en las relaciones de recurrencia. A la vez deben iniciarse con valores las variables involucradas.

```
#define epsilon 1.0e-6
```

```
double reciproco(double a) // 0 < a < 2
{ double x=1., err = fabs(1.-a);
  while (err > epsilon)
  { x = (1+err)*x;
    err = err*err;
  }
  return(x);
}
```

El valor de epsilon corresponde a la exactitud con que se manipulan números flotantes en precisión simple. *No pueden obtenerse más de seis cifras decimales significativas.*

Siempre que se diseña un lazo while, debe verificarse que éste termine. En este caso se requiere verificar que el error va disminuyendo, cada vez que se ejecuta el bloque de repetición dentro del lazo.

De la expresión para el error relativo, puede deducirse una relación para el error en la etapa k -ésima en función del error inicial:

$$e_k = e_{k-1}^2 = (e_{k-2}^2)^2 = ((e_{k-3}^2)^2)^2 = \dots$$

$$e_k = e_{k-1}^2 = e_{k-2}^{2^2} = e_{k-3}^{2^3} = \dots = e_0^{2^k}$$

Es importante destacar el razonamiento inductivo que permite escribir el caso final. En este caso basta observar que la suma del subíndice más el exponente de la potencia de dos, siempre suman k .

Debido a que el exponente es par, para que el error disminuya al aumentar k , debe cumplirse:

$$|e_0| < 1$$

De la expresión del error relativo, el valor inicial debe cumplir:

$$|1 - ax_0| < 1$$

La anterior implica:

$$-1 < 1 - ax_0 < 1$$

Y también:

$$0 < ax_0 < 2$$

Finalmente se obtiene, la condición inicial para garantizar la convergencia:

$$0 < x_0 < \frac{2}{a}$$

Que demuestra la expresión obtenida antes, a partir de la gráfica.

El código anterior, si bien resuelve la dificultad propuesta de no emplear divisiones, no permite obtener respuesta para valores negativos del argumento, ni tampoco entrega resultados para valores del argumento superiores a dos. También puede observarse que no puede tratarse el caso con argumento igual a cero.

La primera modificación del algoritmo básico anterior es la elección de un valor inicial para valores elevados del argumento. Para valores de a mayores que 2, el argumento real puede expresarse:

$$a = f 2^e$$

Con:

$$\frac{1}{2} \leq f < 1$$

El cálculo de e y f , puede lograrse con:

```
f=a; e=0;
while (f >= 1) {e++; f=f/2;}
while (f < 0.5) {e--; f=f*2;}
```

Debe notarse que sólo se efectúa uno de los lazos, y que después de ambos se tiene que se cumple la condición para f .

El recíproco, en términos de e y f :

$$\frac{1}{a} = \frac{1}{f} 2^{-e}$$

Entonces el valor inicial debe cumplir:

$$0 < x_0 < \frac{2}{a} = \frac{2}{f} 2^{-e}$$

Usando el mayor valor posible para f , se tiene:

$$0 < x_0 < 2 \cdot 2^{-e}$$

Entonces un valor seguro inicial que garantice la convergencia es:

$$x_0 = 2^{-e}$$

Y empleando la definición del error relativo, con el valor inicial anterior, se tiene:

$$e_0 = |1 - ax_0| = |1 - f| = 1 - f$$

Entonces $e_0 < 1$, lo cual asegura que el algoritmo converge.

Cuando se trabaja con números flotantes es importante no efectuar comparaciones de igualdad con un valor, dentro de un lazo. Suelen plantearse como la diferencia con el valor menor que una tolerancia. Es decir en lugar de emplear $(x == a)$ debe usarse: $|x - a| < \textit{epsilon}$

Por esta razón se colocan cláusulas condicionales al ingresar a la función. En el diseño evitaremos los casos: $a=1$, $a=0$ y $a=\textit{inf}$.

La extensión para argumentos negativos la trataremos, almacenando el signo del argumento, y calculando el recíproco de números positivos.

```
#define infinito 1./0.
```

```

double reciproco(double a)
{ double x, err, f, p;
  int e, sign;

  if (a==1.) return (1.);
  else if (a==-1.) return (-1.);
  else if (a==0.) return (infinito);
  else if (fabs(a)==infinito ) return (0.);

  if (a<0.) {a = -a; sign=1;} else sign=0;

  if (a<1.)
    {x=1.; err=1.-a;}
  else
    { // a > 1.
      f=a; e=0; p=1; //mediante p se calcula 2^(-e)
      while (f >= 1.) {e++; f=f/2; p=p/2;}
      while (f< 0.5) {e--; f=f*2; p=p*2;}
      x=p; err=1-f;
    }

  while (err>epsilon)
    { x = (1.+err)*x;
      err=err*err;
    }
  if (sign) x=-x;
  return(x);
}

```

Los valores iniciales, para $a > 1$, pueden calcularse con funciones de la biblioteca (math.h), según se ilustra a continuación.

Con: $a = f 2^e$, la función frexp, calcula mantisa ($1/2 \leq f < 1$), y exponente e (de tipo entero).

```

err = 1- frexp(a, &e);
x = pow(2, -e); //valor inicial x= g*2^-e. Tal que: 1 < g <=2

```

Ejercicios propuestos.

E1. Diseñar una función que genera la próxima permutación.

Ahora el arreglo almacena los dígitos desde 0 hasta $(n-1)$. De este modo no puede emplearse el elemento 0 del arreglo como centinela y es preciso modificar el código propuesto antes.

E2. Codificar el siguiente algoritmo, para la raíz cuadrada.

Con: $0 < a < 2$

Para $i=0$

$$x_0 = a$$

$$e_0 = 1 - a$$

Para $i>0$

$$x_i = \left(1 + \frac{e_{i-1}}{2}\right)x_{i-1}$$

$$e_i = e_{i-1}^2 \left(\frac{3}{4} + \frac{e_{i-1}}{4}\right)$$

$$\lim_{i \rightarrow \infty} x_i = \sqrt{a}$$

Debido a que el error relativo puede tener signo, la condición de término de la iteración, puede lograrse con: $fabs(e_i) < epsilon$. Que emplea la función valor absoluto para números flotantes.

Este algoritmo fue implementado en la biblioteca de EDSAC 1, uno de los primeros computadores. El algoritmo sólo utiliza sumas y multiplicaciones.

E3. Codificar el siguiente algoritmo, para la raíz cuadrada.

Comprobar que empleando el método de Newton, se obtiene el algoritmo para calcular la raíz cuadrada, según:

$$x_{k+1} = \frac{1}{2} \left(x_k + \frac{a}{x_k}\right)$$

Converge con: $x_0 > 0$

Referencias.

Edsger W. Dijkstra. "A Short Introduction to the Art of Programming.", 1971.

Índice general.

CAPÍTULO 1.	1
INTRODUCCIÓN A LAS ESTRUCTURAS DE DATOS Y ALGORITMOS.	1
1.1. ESTRUCTURA DE LOS DATOS.	1
1.2. ESTRUCTURA DE LAS ACCIONES. ALGORITMOS.	2
1.3. LENGUAJES.	3
1.4. GENIALIDADES.	4
1.5. TEORÍA Y PRÁCTICA.	4
1.6. DEFINICIONES.	4
1.6.1. <i>Algoritmo</i>	4
1.6.2. <i>Heurística</i>	5
1.6.3. <i>Estructuras de datos</i>	5
1.7. ALGORITMOS CLÁSICOS.	5
1.7.1. <i>Algoritmo de Euclides</i>	6
1.7.2. <i>Algoritmo el colador de Erastótenes</i>	8
1.7.3. <i>Permutaciones en orden lexicográfico. Algoritmo de Dijkstra (1930-2002)</i>	9
PROBLEMAS RESUELTOS.	11
P1. <i>Combinaciones en orden lexicográfico</i>	11
P2. <i>Cálculo del recíproco</i>	15
EJERCICIOS PROPUESTOS.	19
E1. <i>Diseñar una función que genera la próxima permutación</i>	19
E2. <i>Codificar el siguiente algoritmo, para la raíz cuadrada</i>	20
E3. <i>Codificar el siguiente algoritmo, para la raíz cuadrada</i>	20
REFERENCIAS.	20
ÍNDICE GENERAL.	21
ÍNDICE DE FIGURAS.	21

Índice de figuras.

FIGURA 1.1. ESTRUCTURAS DE DATOS.	1
FIGURA 1.2. ESTRUCTURA DE ACCIONES.	2
FIGURA 1.3. SECUENCIA DE APROXIMACIÓN USANDO EL MÉTODO DE LA TANGENTE.	15