PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ
FACULTAD DE CIENCIAS E INGENIERÍA

Algoritmia
Tema para el laboratorio 2

Estrategias algorítmicas:
Divide y vencerás, Programación dinámica

Programación dinámica

El problema de la fila de monedas. Hay una fila de n monedas cuyos valores son enteros positivos $c_1, c_2, \ldots, c_n$, no necesariamente diferentes. El objetivo es encontrar la máxima cantidad de dinero sujeto a la restricción de que no se pueden tomar dos monedas adyacentes en la fila inicial.

Sea $F(n)$ la cantidad máxima que puede ser tomada de la fila de n monedas. Para obtener una fórmula recursiva para $F(n)$, dividimos todas las selecciones permitidas de monedas en dos grupos: aquella que incluye la última moneda y aquella que no la incluye. La cantidad máxima que se puede obtener del primer grupo es $c_n + F(n-2)$ – el valor de la enésima moneda más la cantidad máxima que podemos tomar de los primeros $n-2$ monedas. La cantidad máxima que podemos obtener del segundo grupo es igual a $F(n-1)$ por la definición de $F(n)$. Por tanto, se tiene la siguiente fórmula recursiva sujeta a las condiciones iniciales:

$$F(n) = \max\{c_n + F(n-2), F(n-1)\} \quad \text{para} \quad n > 1,$$
$$F(0) = 0, \quad F(1) = c_1.$$

Luego, se puede calcular $F(n)$ llenando una tabla de una sola fila de izquierda a derecha, tal como se muestra en el siguiente ejemplo, donde $C[6] = \{5,1,2,10,6,2\}$ y el algoritmo nos conduce a $F(6) = 17$.

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

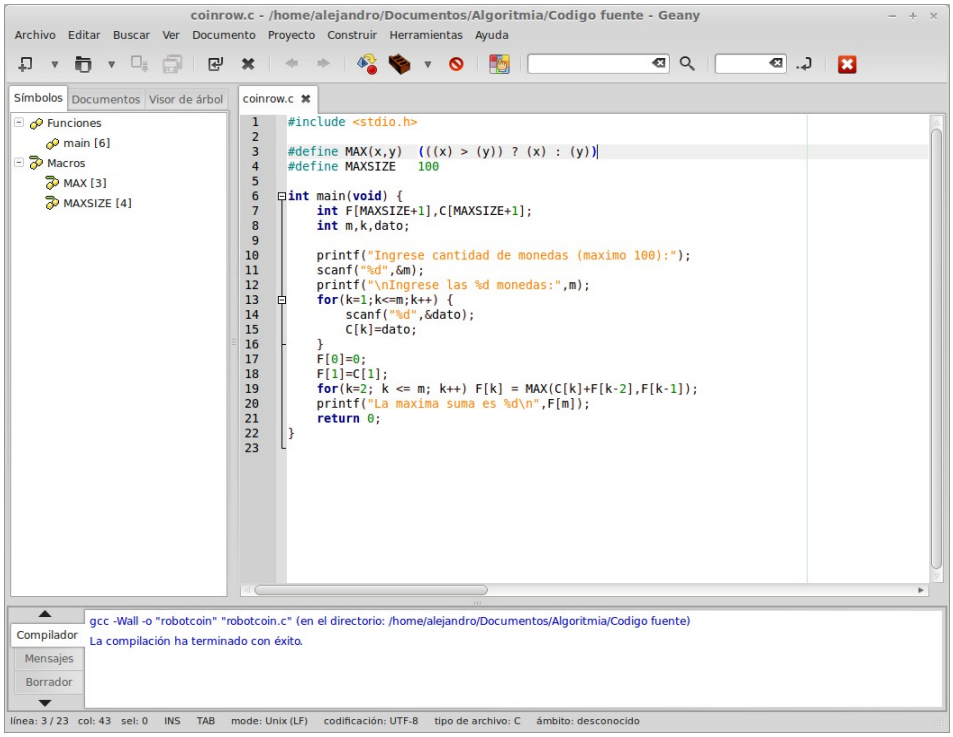
index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

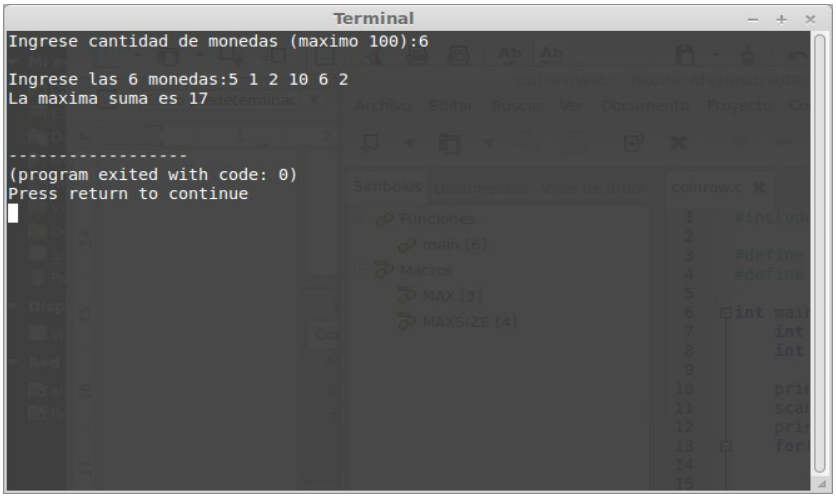
index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Para encontrar las monedas con el valor total máximo encontrado, se necesita rastrear en retroceso los cálculos para ver cuál de las dos posibilidades $-c_n + F(n-2)$ ó $F(n-1)$– origina el máximo en $F(n)$. En el ejemplo, el último cálculo de la fórmula mostró que el máximo fue la suma $c_6 + F(4)$, lo que significa que la moneda c_6=2 es una parte de la solución óptima. Si seguimos retrocediendo, para calcular $F(4)$, el valor máximo fue originado por la suma $c_4 + F(2)$, lo que significa que la moneda c_4=10 es parte de la solución óptima también. Finalmente, el máximo en el cálculo de $F(2)$ fue producido por $F(1)$, lo que implica que la moneda c_2 no es parte de la solución óptima y la moneda c_1=5 si lo es. Por tanto la solución óptima es $\{c_1, c_4, c_6\}$. Para evitar repetir los mismos cálculos durante el rastreo en retroceso, la información sobré cuál de los dos términos es mayor, puede ser registrado en un arreglo extra cuando los valores de F son calculados.

A continuación el programa



y su ejecución



El problema del cambio de monedas. Considere el caso general del siguiente bien conocido problema. Proporcionar un número mínimo de monedas de denominaciones $d_1 < d_2 < \ldots < d_m$ de forma que el total sea igual a una cantidad n. Se asume que hay una disponibilidad ilimitada de cantidad de monedas por cada una de las m denominaciones $d_1 < d_2 < \ldots < d_m$ donde $d_1 = 1$.

Acordemos en denominar $F(n)$ al número mínimo de monedas cuyo valor suman n; es conveniente definir $F(0) = 0$. La cantidad n solo puede ser obtenida añadiendo una moneda de denominación d_j a la cantidad $(n - d_j)$ para $j = 1, 2, \ldots, m$ tal que $n \geq d_j$. Además se puede considerar todas estas denominaciones y seleccionar las monedas minimizando $F(n - d_j) + 1$. Donde 1 es una constante, por supuesto, se puede calcular primero el mínimo de $F(n - d_j)$ y luego sumarle 1. De aquí se tiene la siguiente fórmula recursiva $F(n)$:

$$F(n) = \min_{j:\, n \geq dj} \{F(n - d_j)\} + 1 \quad \text{para} \quad n > 0$$

$$F(0) = 0$$

Se puede calcular $F(n)$ llenando una tabla de una fila de izquierda a derecha de manera similar a como se realizó en el problema anterior, la diferencia estriba que para calcular la entrada de la tabla se requiere hallar el mínimo de hasta m números.

La ejecución del algoritmo para una cantidad $n = 6$ y con monedas de las siguientes denominaciones 1, 3 y 4 se muestran a continuación.

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1 - 1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2 - 1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3 - 1], F[3 - 3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4 - 1], F[4 - 3], F[4 - 4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5 - 1], F[5 - 3], F[5 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

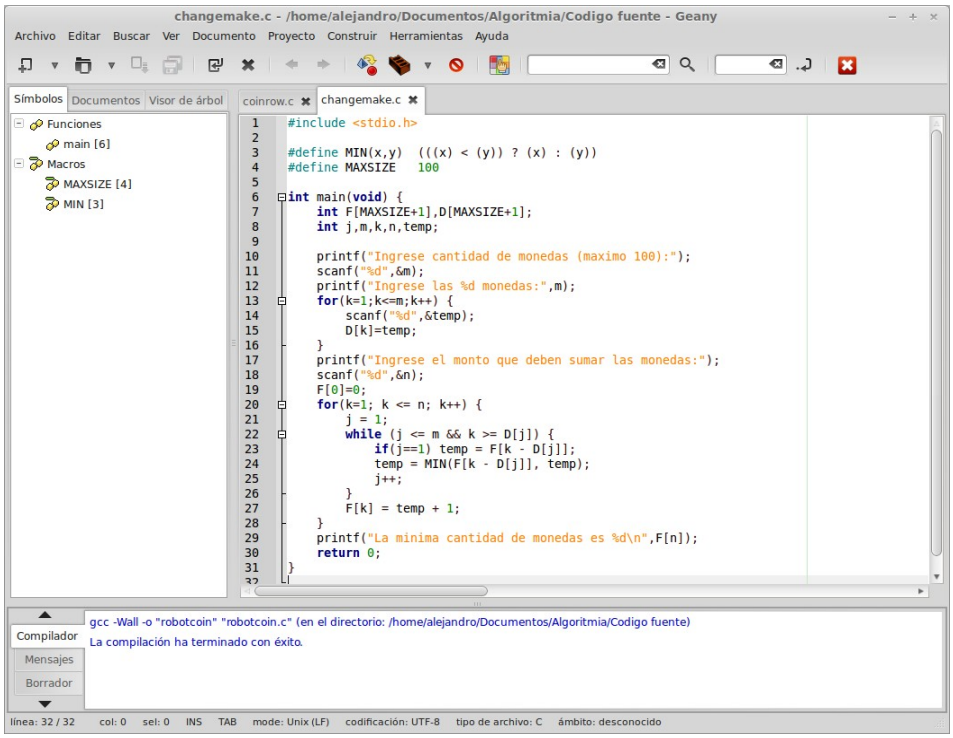
$F[6] = \min\{F[6 - 1], F[6 - 3], F[6 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

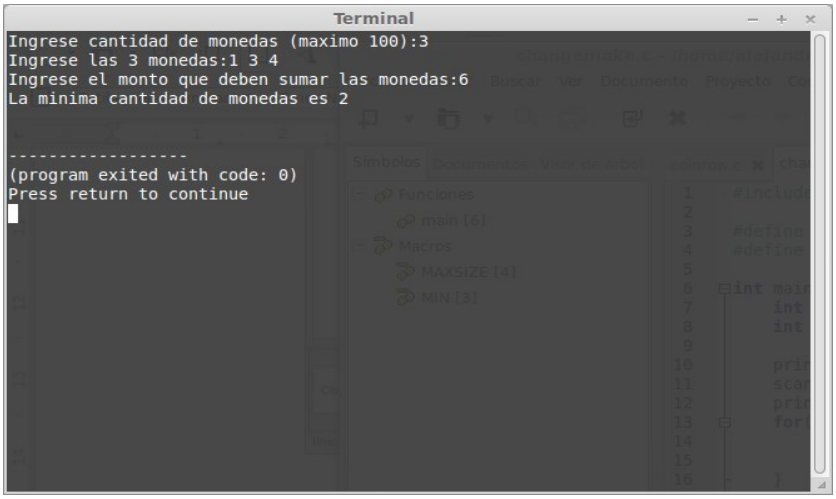
La eficiencia en tiempo es $O(nm)$ y la eficiencia en espacio es $\Theta(n)$.

Para encontrar las monedas de una solución óptima, se necesita seguir el rastreo en retroceso de los cálculos para ver cuál de las denominaciones produjo el valor mínimo en la fórmula obtenida. Por ejemplo considere, la última aplicación de la fórmula (para $n = 6$), el valor mínimo fue producido por $d_2 = 3$. El segundo mínimo (para $n = 6 - 3$) también fue producida por una moneda de esa denominación. Por tanto, el conjunto mínimo de monedas para $n = 6$ son dos monedas de 3.

A continuación el programa:



y su ejecución



El problema de la recolección de monedas. Muchas monedas son colocadas en las casillas de un tablero de n x m, no más de una moneda por cada casilla. Un robot, ubicado en una casilla de la parte superior izquierda del tablero, necesita recoger tantas monedas como le sea posible y llevarlas a la casilla de la parte inferior derecha. En cada paso el robot puede moverse una casilla a la derecha o una casilla hacia abajo de su actual posición. Cuando el robot visita una casilla con una moneda, siempre recoge la moneda. Diseñe un algoritmo para encontrar el máximo número de monedas que el robot puede recoger y la ruta que debe seguir para lograr este objetivo.

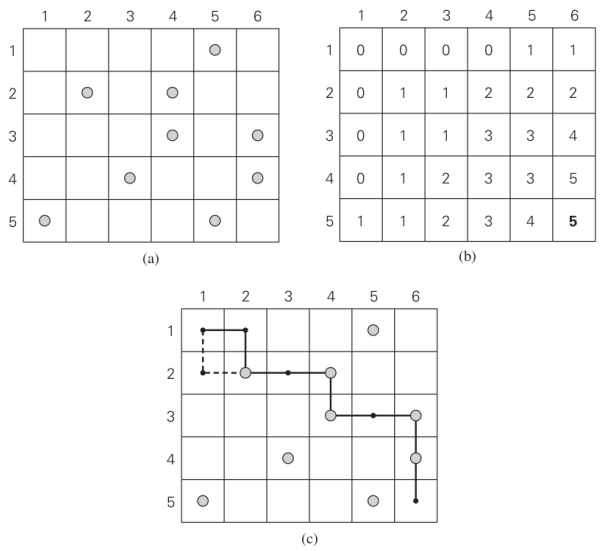
Tomemos como $F(i, j)$ al mayor número de monedas que el robot puede recoger y traer a la casilla (i, j) en la i-ésima fila y la j-ésima columna del tablero. El robot puede alcanzar este casilla ya sea desde la casilla adyacente de arriba $(i-1, j)$ o desde la casilla adyacente de la izquierda $(i, j-1)$. El número mayor de monedas que puede ser recogidas a estas celdas son $F(i-1, j)$ y $F(i, j-1)$, respectivamente. Por supuesto, no hay casillas adyacentes de arriba en la primera fila de casillas, y no hay casillas adyacentes a la izquierda en la primera columna de casillas. Para estas casillas asumimos que $F(i-1, j)$ y $F(i, j-1)$ son igual a 0 debido a que no existen vecinos. Por tanto, el mayor número de monedas que el robot puede recoger a la casilla (i, j) es el máximo de estos dos números más una posible moneda en la misma casilla (i, j). En otras palabras, tenemos la siguiente fórmula para $F(i, j)$:

$$
\begin{array}{ll}
F(i,j) = \max\{F(i-1,j), F(i,j-1)\} + c_{ij} & \text{para } 1 \le i \le n,\ 1 \le j \le m \\
F(0,j) = 0 & \text{para } 1 \le j \le m \\
F(i,0) = 0 & \text{para } 1 \le i \le n,
\end{array}
$$

donde $c_{ij} = 1$ si hay una moneda en (i, j) y $c_{ij} = 0$ en caso contrario.

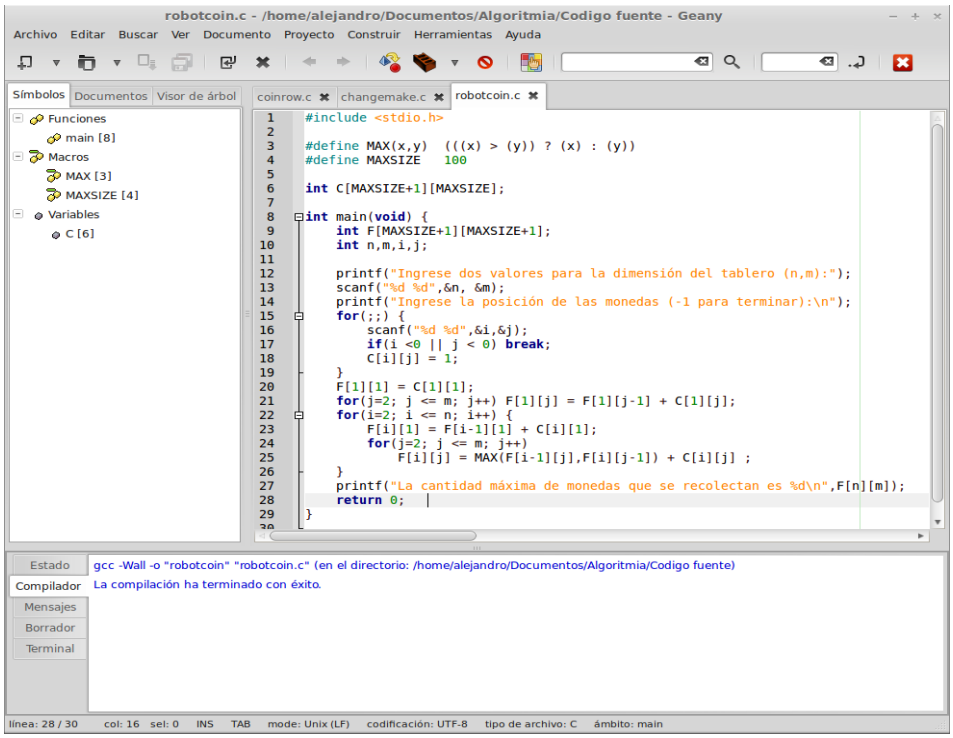
Usando estas fórmulas, se puede llenar el tablero n x m de los valores $F(i, j)$ ya sea fila por fila o columna por columna, como es típico en algoritmos de programación dinámica emplear arreglos de dos dimensiones.

A continuación se muestra el arreglo obtenido (b) a partir de la distribución inicial ofrecida (a). Donde calcular el valor de $F(i, j)$, empleando la fórmula obtenida arriba, para cada casilla del tablero toma un tiempo constante, su eficiencia en tiempo es $\Theta(nm)$ y su eficiencia en espacio también es $\Theta(nm)$.

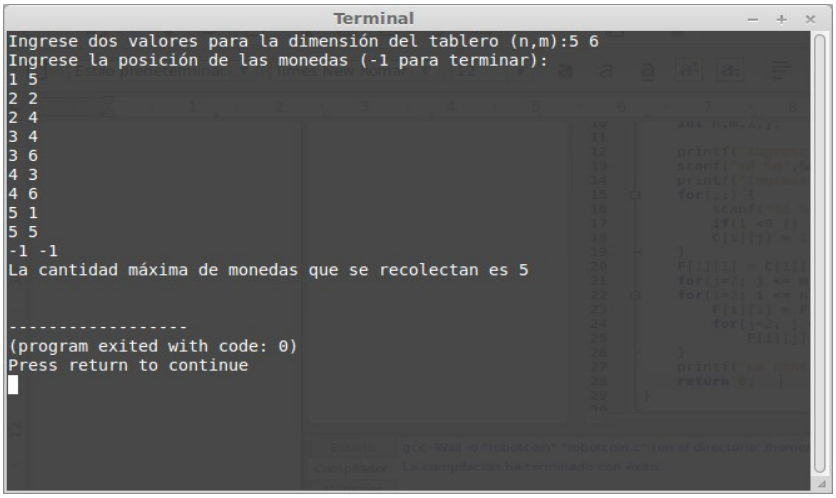


Rastreando en retroceso los cálculos es posible obtener la ruta óptima de la siguiente forma: si $F(i-1, j) > F(i, j-1)$, significa que una ruta óptima para la casilla (i, j) debe venir de la casilla superior adyacente; si $F(i-1, j) < F(i, j-1)$, significa que una ruta óptima a la casilla (i, j) debe venir de la casilla izquierda adyacente; y si $F(i-1, j) = F(i, j-1)$, significa que se puede alcanzar la casilla (i, j) desde cualquier dirección. Para el ejemplo mostrado arriba, la fórmula conduce a dos rutas óptimas para el caso presentado en (a), estas dos soluciones son presentados en (c).

A continuación el programa:



y su ejecución



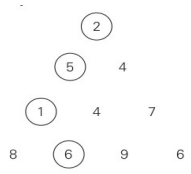
Ejercicios de programación dinámica

1.- Modifique el programa *coinrow.c* para que muestre no solo la suma máxima, sino cuáles son los valores que me permiten obtener la suma máxima.

2.- Modifique el programa *changemake.c* para no sólo muestre la cantidad de mínima de monedas que suman la cantidad ingresada, sino de qué denominación son dichas monedas.

3.- El problema del corte de una barra. Escriba un programa en C que empleando programación dinámica resuelva el siguiente problema. Encontrar el máximo del precio de venta total que se puede obtener de cortar una barra de n unidades de largo en piezas de longitud entera si el precio de venta de una pieza de i unidades es p_i para $i = 1,2, \ldots, n$.

4.- El problema de la mínima suma descendente. Ciertos enteros positivos son colocados en un triángulo equilátero con n números en su base, tal como se muestra abajo para $n = 4$. El problema es encontrar la suma más pequeña en forma descendente desde el vértice del triángulo a su base, a través de una secuencia de números adyacentes. Escriba un programa en C, que aplicando la estrategia de programación dinámica resuelva este problema.



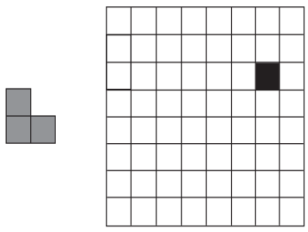
5.- Coeficiente binomial. Escriba un programa en C empleando la estrategia de programación dinámica que calcule el coeficiente binomial $C(n, k)$ el programa no debe emplear multiplicaciones.

Ejercicio de Divide y Vencerás.

6.- Escriba un programa en C que aplique la estrategia divide y vencerás para encontrar la posición del mayor elemento en un arreglo de n números.

7.- Escriba un programa en C que aplique la estrategia divide y vencerás para calcular a^n donde n es un entero positivo.

8.- Un *tromino* (más exactamente conocido como *tromino* derecho) es un mosaico de forma de L formado por tres cuadrados de 1 x 1. El problema es cubrir cualquier tablero de ajedrez de $2n$ x $2n$ con un cuadrado perdido (que no se debe considerar) empleando solo *trominos*. Los *trominos* pueden ser orientados de cualquier forma, pero ellos deben de cubrir todos los cuadrados del tablero excepto el cuadrado que está perdido y los *trominos* no se deben superponer.



Escriba un programa en C que aplicando la estrategia devide y vencerás resuelva el problema del *tromino* arriba expuesto.

9.- A continuación se detalla un algoritmo para multiplicar dos números enteros grandes, por ejemplo de 20 cifras. Lea, analice e implemente un programa en C que aplique dicho algoritmo para obtener el producto de dos números grandes.

Multiplication of Large Integers

Some applications, notably modern cryptography, require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment. This practical need supports investigations of algorithms for efficient manipulation of large integers. In this section, we outline an interesting algorithm for multiplying such numbers. Obviously, if we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications. (If one of the numbers has fewer digits than the other, we can pad the shorter number with leading zeros to equalize their lengths.) Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, this turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, say, 23 and 14. These numbers can be represented as follows:

$$23 = 2 \,.\, 10^1 + 3 \,.\, 10^0 \quad \text{and} \quad 14 = 1 \,.\, 10^1 + 4 \,.\, 10^0 \,.$$

Now let us multiply them:

$$23 * 14 = (2 \,.\, 10^1 + 3 \,.\, 10^0) * (1 \,.\, 10^1 + 4 \,.\, 10^0)$$

$$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0 \,.$$

The last formula yields the correct answer of 322, of course, but it uses the same four digit multiplications as the pen-and-pencil algorithm. Fortunately, we can compute the middle term with just one digit multiplication by taking advantage of the products $2 * 1$ and $3 * 4$ that need to be computed anyway:

$$2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - 2 * 1 - 3 * 4.$$

Of course, there is nothing special about the numbers we just multiplied. For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula

$$c = a * b = c_2\, 10^2 + c_1\, 10^1 + c_0 \,,$$

where

$c_2 = a_1 * b_1$ is the product of their first digits,
$c_0 = a_0 * b_0$ is the product of their second digits,
$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's digits and the sum of the b's digits minus the sum of c_2 and c_0 .188

Now we apply this trick to multiplying two n-digit integers a and b where n is a positive even number. Let us divide both numbers in the middle—after all, we promised to take advantage of the divide-and-conquer technique. We denote the first half of the a's digits by a_1 and the second half by a_0 ; for b, the notations are b_1 and b_0 , respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$$c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$$

$$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$$

$$= c_2 10^n + c_1 10^{n/2} + c_0 ,$$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the a's halves and the sum of the b's halves minus the sum of c_2 and c_0 .

If $n/2$ is even, we can apply the same method for computing the products c_2 , c_0 , and c_1 . Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

Prof. Alejandro T. Bello Ruiz