Instituto Politécnico Nacional

Escuela Superior de Cómputo

Análisis de Algoritmos

Ejercicios 02

Determinando funciones de complejidad

Alumno:
Luis Fernando RESÉNDIZ CHÁVEZ

24 de octubre de 2020

1. Código 01

1.1. Código implementado por el profesor

```
1  void function() {
2      int n, i, j, temp;
3      scanf("%d", &n);
4
5      for(i = 1; i < n; ++i)
6          for(j = n; j > 1; j /= 2) {
7              temp = A[j];
8              A[j] = A[j + 1];
9              a[j + 1] = temp;
10         }
11 }
```

1.2. Análisis del código

Primero vamos a evaluar todas las operaciones que se realizan en cada for, comenzando con el primer for en la linea 5, las iteraciones van desde $i = 1$ hasta $n - 1$, significa que todas las operaciones que se hagan dentro de este for, se harán mínimo $n - 2$ veces, comenzando con la asignación de la i, ésta solo se hará una vez, posteriormente, las comparaciones que tiene que hacer para seguir ejecutando el for y después salir de el, serán $n - 1$ veces. El incremento de i se toma como dos acciones, la asignación y la suma de una unidad, la complejidad total de éste for es: $(2n - 2) + 1$.
Continuando con el siguiente for, en la linea 6, la variable j inicia en n, y se ejecuta mientras $j > 1$, hay que notar que el decremento de la variable j es ir dividiéndola entre dos, en otras palabras, la complejidad es $\log_2 n$, le sumamos dos unidades más, por la asignación del principio y otra por la comparación extra para salir del for. La complejidad de éste for es: $4 \log_2(n) + 2$.
Dentro de éste for, tenemos mas operaciones que contar, en la linea 7 hay una asignación y una consulta al un arreglo. En la siguiente linea, hay dos consultas al arreglo, una asignación y una operación aritmética, es decir, 4 operaciones. En la siguiente linea tenemos una consulta al arreglo, una suma y una asignación, 3 operaciones.
Finalmente podemos decir que estas operaciones se van a multiplicar por el numero de veces que se repita su for padre, y asu vez el for padre se va a a repetir las iteraciones de su for padre.

La complejidad espacial esta dad por el tamaño del arreglo, que en este caso vamos a suponer que es de tamaño n, y además se ocupan las varibales $i, j, temp$, por lo tanto la complejidad espacial es $n+3$.

```
1 for(i = 1; i < n; i = i + 1) // 3*n-2 + 2
2     for(j = n; j > 1; j = j / 2) { // 4*log(n) + 2
3         temp = A[j]; // 2
4         A[j] = A[j + 1]; // 4
5         A[j + 1] = temp; // 3
6     }
```

1.3. La función complejidad temporal f(n)

$$f(n) = ((3n - 2) + 1) * ((13 \log_2 n) + 1) \tag{1}$$

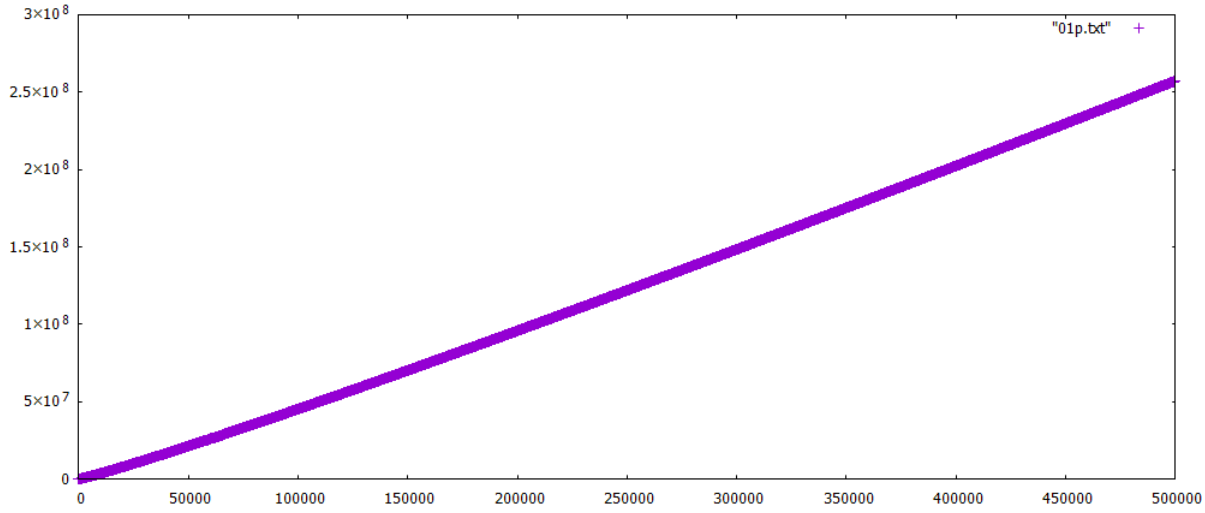
1.4. La función complejidad espacial f(n)

$$f(n) = n + 3 \tag{2}$$

1.5. Tabla comparativa de la función complejidad

n	Conteo Teórico	Conteo Empírico
-1	0	0
0	0	0
1	2	2
2	50	50
3	122	122
5	306	306
15	1593	1593
20	2356	2356
100	18199	18199
409	97072	97072
500	122603	122603
593	149365	149365
1000	272311	272311
1471	422749	422749
1500	432227	432227
2801	875408	875408
3000	945639	945639
5000	1675740	1675740
10000	3621912	3621912
20000	7784590	7784590

1.6. Gráfica muestral sobre 500,000 puntos



1.6.1. Código de la función complejidad f(n)

```
int f(int n) {
    return (n <= 0) ? 0 : ((3*n-2) + 1) * ((13*log(n)) + 1);
}
```

2. Código 02

2.1. Código implementado por el profesor

```
int polinomio = 0;
for(i = 0; i <= n; i = i + 1) {
    polinomio = polinomio * z + A[n-i];
}
```

2.2. Análisis del código

Para calcular la función complejidad, podemos empezar con la primer linea de código, la cual solante tiene una asignación de una variables.
Después nos encontramos con un for, el cual es controlado por la variable i, empieza desde cero y termina mientras la i sea menor o igual que n y el incremento es de uno en uno, por lo tanto la compljidad inicial es $n+1$, despues hat que contar las primer asignación de la variable i, la comparación extra para salir del for, y las asignaciones y sumas al incrementar la i, por lo tanto la función principal queda: $(3n+1)+2$.
Después falta caluclar las cosas que suceden dentro del for, las cuales son cinco: la asignación al polinomio, la multiplicaión por z, la suma con un elemento del arreglo, la resta de la posición del arreglo y la consulta al arreglo.

```
int polinomio = 0; // 1
for(i = 0; i <= n; i = i + 1) { // (3*n+1) + 2
    polinomio = polinomio * z + A[n-i]; // 5
}
```

2.3. La función complejidad temporal f(n)

$$f(n) = (8n + 1) + 2 \tag{3}$$

2.4. La función complejidad espacial f(n)

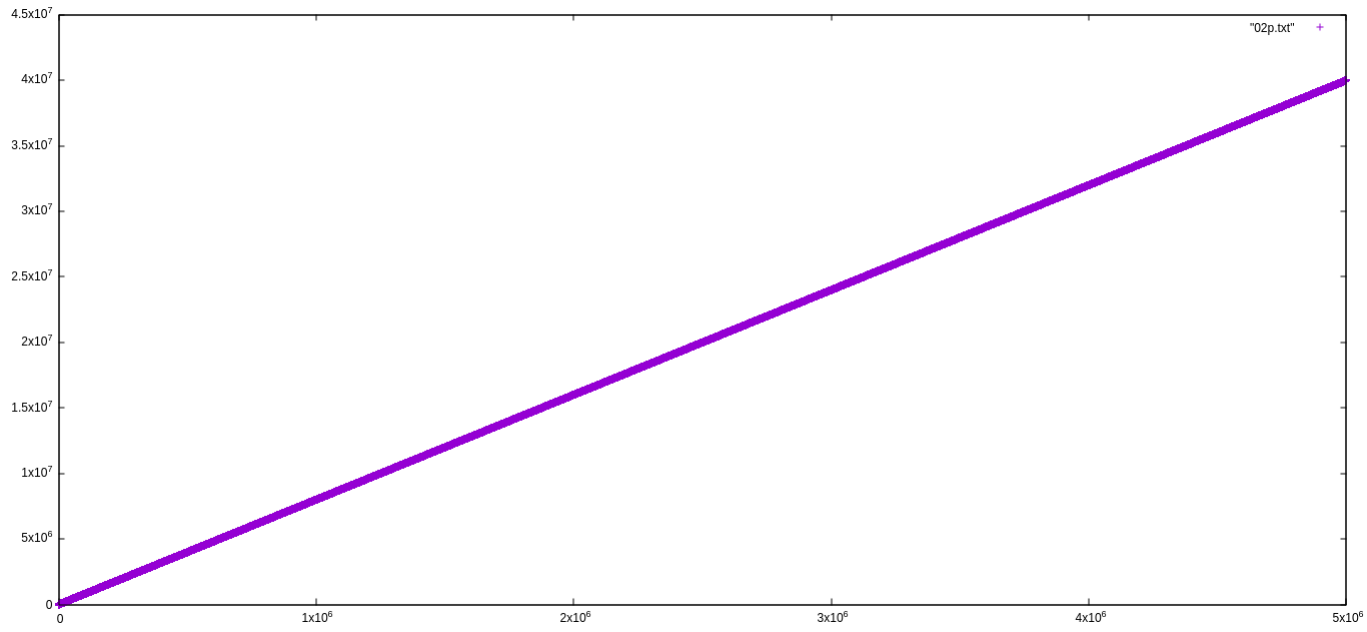
$$f(n) = n + 4 \tag{4}$$

Cuadro 1: Tabla comparativa del código 1

n	Conteo Teórico	Conteo Empírico
1	18	18
0	10	10
1	18	18
2	26	26
3	34	34
5	50	50
15	130	130
20	170	170
100	810	810
409	3282	3282
500	4010	4010
593	4754	4754
1000	8010	8010
1471	11778	11778
1500	12010	12010
2801	22418	22418
3000	24010	24010
5000	40010	40010
10000	80010	80010
20000	160010	160010
20000	160010	160010

2.5. Tabla comparativa de la función complejidad

2.6. Gráfica muestral sobre 500,000 puntos



2.6.1. Código de la función complejidad f(n)

```
int f(int n) {
    return (8*(n+1)) + 2;
}
```

3. Código 03

3.1. Código implementado por el profesor

```
for(i = 1; i <= n; ++i) {
    for(j = 1; i <= n; ++j) {
        c[i][j] = 0;
        for(k = 1; k <= n; ++k) {
            c[i][j] = c[i][j] + a[i, k] * b[k][j];
        }
    }
}
```

3.2. Análisis del código

El primer for, es un for lineal que va desde 1 hasta n, por lo tanto todo lo que suceda dentro se hará n veces, entonces podemos observar que va a hacer n comparaciones para continuar en el for y una más para salir de el, tambien hrá n asigaciones y n sumas a la variable que lo controla, en este caso es i. No olvidemos la primer asignación de $i = 1$, en total, la complejidad de este for es:

$$3n + 2 \tag{5}$$

En el segundo for sucede exactamente lo mismo que en el for anterior, solo que esta vez se va a repetir las veces que dicta el primero:

$$3n(3n + 2) + 2 \tag{6}$$

Dentro del for se ejecuta una instrucción la cual tiene una asignación y una consulta a una matriz, entonces la complejidad del for aumenta:

$$3n(5n + 2) + 2 \tag{7}$$

En el ultimo for, suecede exactamente lo mismo que en los ateriores, por lo tanto se modifica la ecuacion, quedando de la siguiente manera:

$$3n(5n(3n + 2) + 2) + 2 \tag{8}$$

Y finalmente agregamos las instrucciones que están dentro del tercer for, las 4 consultas a las diferentes matrices, la suma, la multiplicación y la asignación, por lo tanto la ecuación queda:

$$3n(5n(10n + 2) + 2) + 2 \tag{9}$$

Resolviendo la ecuación:

$$3n(5n(10n + 2) + 2) + 2 \tag{10}$$

$$3n(50n^2 + 10n + 2) + 2 \tag{11}$$

$$150n^3 + 30n^2 + 6n + 2 \tag{12}$$

Ahora, para calcular el espacio de memoria, hay que saber las dimensiones de la matriz, como todos los arreglos llegan hasta n, voy s suponer que mínimo las matrices son de tamaño $(n+1)*(n+1)$, como son 3, el resultado se multiplica por 3. Las variables que iteran entre los for y acceden a las matrices también son 3, por lo tanto la complejidad espacial es: $3(n^2 + 2n + 1) + 3$

3.3. La función complejidad temporal f(n)

$$f(n) = 150n^3 + 30n^2 + 6n + 2 \tag{13}$$

3.4. La función complejidad espacial f(n)

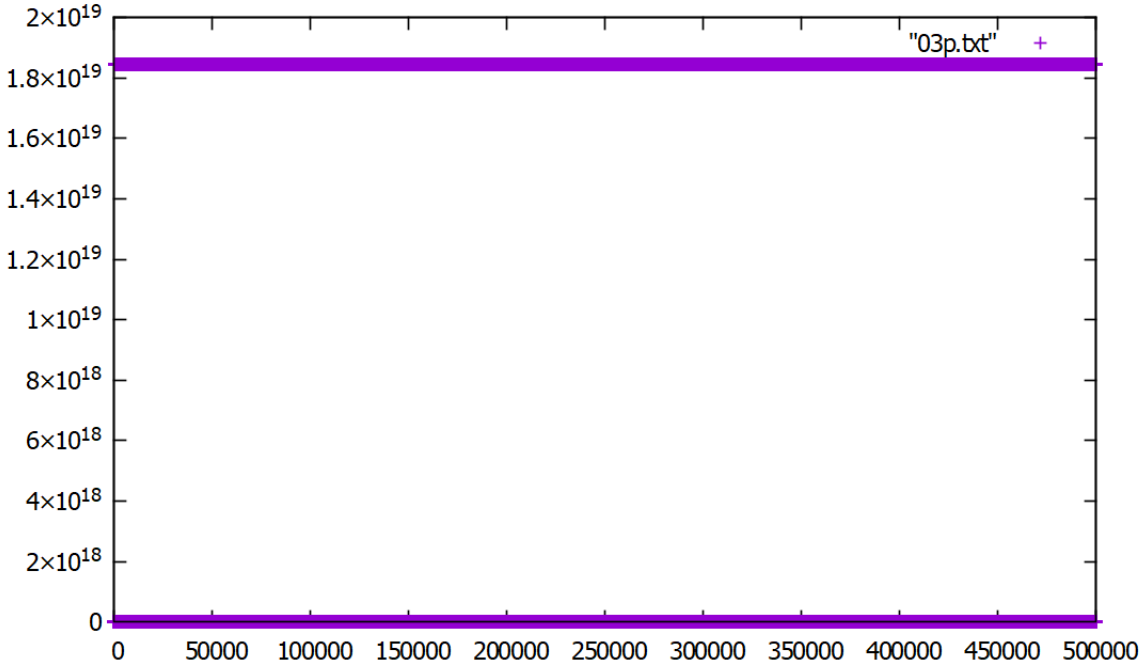
$$f(n) = 3(n^2 + 2n + 1) + 3 \tag{14}$$

3.5. Tabla comparativa de la función complejidad

Cuadro 2: Tabla comparativa del código 1

n	**Conteo Teórico**	**Conteo Empírico**
-1	4201083	0
0	8594135675	8589934592
1	807458052731	807453851648
2	5729490573947	5729486372864
3	18640162265723	18640158064640
5	83889305426555	83889301225472
15	2203713364040315	2203713359839232
20	5206024352963195	5206024348762112
100	645536170163313275	645536170159112192
409	7205991521009539707	7205991521005338624
500	6775885653377817211	6775885653373616128
593	5261165320755747451	5261165320751546368
1000	17184670691153156731	17184670695443922943
1471	3328903961427057275	3328903961422856192
1500	16348085931762260603	16348085936053026815
2801	9984634838088424059	9984634842379190271
3000	497605779818289787	497605779814088704
5000	13820272587420211835	13820272591710978047
10000	5442785288541706875	5442785288537505792
20000	10447872581657696891	10447872585948463103

3.6. Gráfica muestral sobre 500,000 puntos



3.6.1. Código de la función complejidad f(n)

```
#define ull unsigned long long int

// f(n)=150n^3 + 30n^2 + 6n + 2
ull f(int n) {
    return (n < 0) ? 0 : ((150 * n * n * n) + (30 * n * n) + (6 * n) + 2);
}
```

4. Código 04

4.1. Código implementado por el profesor

4.2. Análisis del código

Comenzamos con la asignación de dos variables, la complejidad hasta el momento es 2, entramos al while siempre y cuando la variable n sea mayor que dos, dentro del for, la n se decrementa en 1, es decir el while hará $n-2$ comparaciones para entrar y una más para salir, la complejidad es: $n-2+1 = n-1$. Dentro del while se hacen 4 asignaciones, una suma y una resta, por lo tanto la complejidad de estas operaciones es 6 por el numero de veces que entra el while.

$$6(n-2)+1 \tag{15}$$

$$6n-12+1 \tag{16}$$

$$6n-11 \tag{17}$$

Para calcular la complejidad espacial hay que contar las variables que se usaron, en este caso son 4, y no importa que valor de n sea, siempre serán 4.

4.3. La función complejidad temporal f(n)

$$f(n) = \begin{cases} 0 & si \quad n < 2 \\ \\ 6n-11 & si \quad n \geq 2 \end{cases} \tag{18}$$

4.4. La función complejidad espacial f(n)

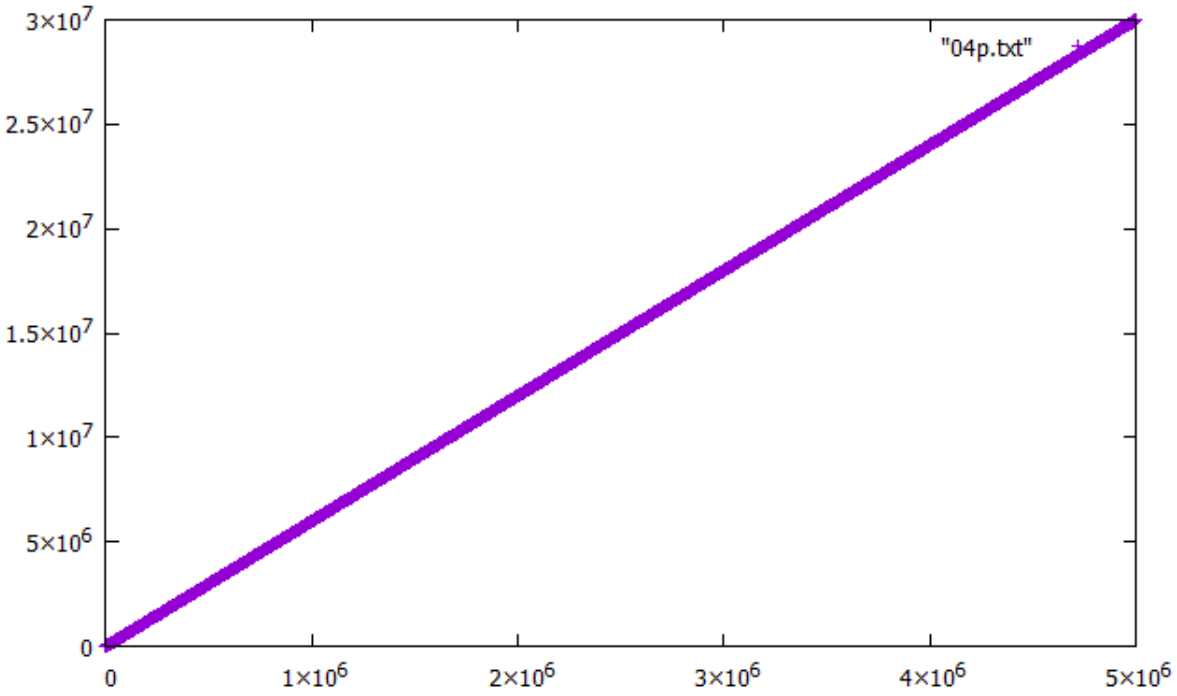
$$f(n) = 4 \tag{19}$$

Cuadro 3: Tabla comparativa del código 1

n	Conteo Teórico	Conteo Empírico
-1	0	0
0	0	0
1	0	0
2	1	1
3	7	7
5	19	19
15	79	79
20	109	109
100	589	589
409	2443	2443
500	2989	2989
593	3547	3547
1000	5989	5989
1471	8815	8815
1500	8989	8989
2801	16795	16795
3000	17989	17989
5000	29989	29989
10000	59989	59989
20000	119989	119989

4.5. Tabla comparativa de la función complejidad

4.6. Gráfica muestral sobre 500,000 puntos



4.6.1. Código de la función complejidad f(n)

```
int f(int n) {
    return (n < 2) ? 0 : 6 * n - 11;
}
```

5. Código 05

5.1. Código implementado por el profesor

```
for(i = n - 1; j = 0; i >= 0; i--, j++) {
    s2[j] = s[i];
}

for(k = 0, k < n; k++) {
    s[i] = s2[i];
}
```

5.2. Análisis del código

Si fueramos estrictos, el código esta mal escrito, po lo tanto no se podría calcular la complejidad. Sin embargo, se entiende lo que el profesor quería hacer y el código correcto es:

```
for(i = n - 1, j = 0; i >= 0; i--, j++) {
    s2[j] = s[i];
}

for(k = 0; k < n; k++) {
    s[i] = s2[i];
}
```

Una vez teniendo el código correcto, en el primer for nos damos cuenta que hay dos variables i y j, sin embargo para que ese for se siga cumpliendo solamente se evalua una, en este caso es la variable i, como la i va disminuyendo de uno en uno desde n-1 hasta 0, entonces las comparaciones que se harán para que entre el for serán n más una extra cuando se da cuenta que ya no es válido. Al principio a la variable i se le asigna un valor el cual se disminuye en 1, por lo tanto hay 2 operaciones más, $n+3$, a la variable j se le asigna el 0, entonces: $n+4$, finalmente al actualizar las variables, se hacen 2 operaciones en cada una, la complejidad aumenta: $4n-4$.
Dentro de ese for, se realiza una operacion que involucra la consulta de 2 arreglos y una asignación, por lo tanto la complejidad final para éste primer arreglo es:

$$7n - 4 \tag{20}$$

Para el segundo arreglo, va desde 0 hasta $n-1$ con incremento de 1, por lo tanto tambien se harán n comparaciones mas una extra para terminar el for, se suma la asignacion de 0 a la variable k y se multiplica por las dos operaciones de adición y asignacion al actualizarla:

$$3n + 2 \tag{21}$$

Dentro del for, se realiza una operación que involucra la consulta de 2 arreglos y una asignación, por lo tanto la complejidad final para éste segundo arreglo es:

$$6n + 2 \tag{22}$$

La función que calcula la complejidad dada una n es la suma de mabs complejidades de los for.
Para calcular la complejidad espacial asumiré que el tamaño de ambos arreglos será de n, como son dos, sería $2n$ más las 4 variables utilizadas i, j, k, n.

5.3. La función complejidad temporal f(n)

$$f(n) = 9n + 4 \tag{23}$$

5.4. La función complejidad espacial f(n)

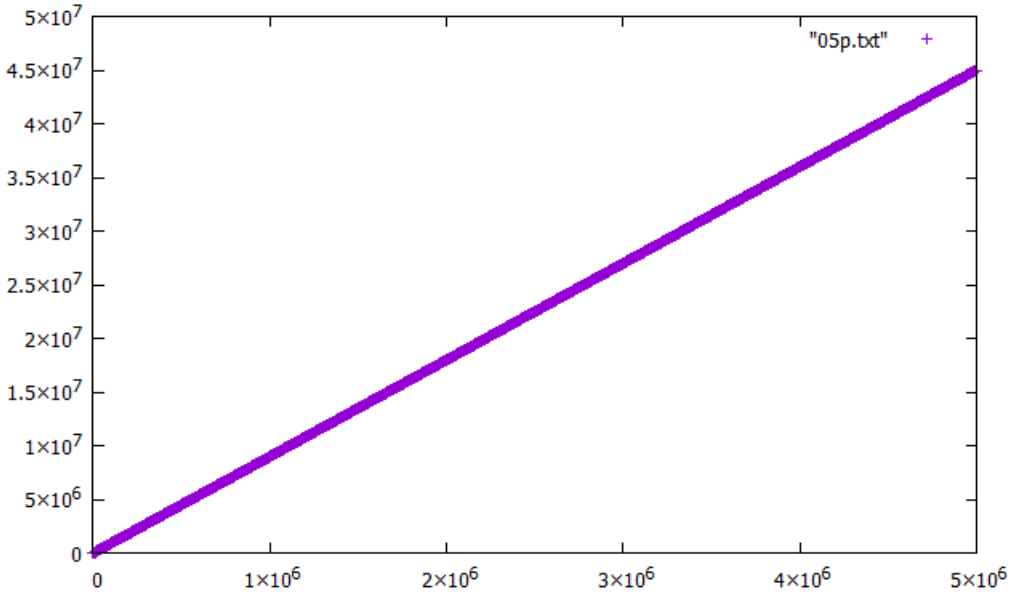
$$f(n) = 2n + 4 \tag{24}$$

5.5. Tabla comparativa de la función complejidad

Cuadro 4: Tabla comparativa del código 1

n	**Conteo Teórico**	**Conteo Empírico**
-1	-5	-5
0	4	4
1	13	13
2	22	22
3	31	31
5	49	49
15	139	139
20	184	184
100	904	904
409	3685	3685
500	4504	4504
593	5341	5341
1000	9004	9004
1471	13243	13243
1500	13504	13504
2801	25213	25213
3000	27004	27004
5000	45004	45004
10000	90004	90004
20000	180004	180004

5.6. Gráfica muestral sobre 500,000 puntos



5.6.1. Código de la función complejidad f(n)

```
int f(int n) {
    return 9 * n + 4;
}
```

6. Código 06

6.1. Código implementado por el profesor

```
void code() {
    l = (a < b) ? a : b;
    r = 1;
    for(i = 2; i<=l; ++i) {
        if(a%i == 0 && b%i == 0) {
            r = i;
        }
    }
}
```

6.2. Análisis del código

Iniciando con la primer linea, existe un operador ternario que le asigna un valor a la variable l, en éste operador existe una comparación y una asignación, en la siguiente linea está una asignación a la variable r, en total hay 3 operaciones.
Es un for donde la variable i va desde 2 hasta l, con aumento de uno en uno, por lo tanto hara $l-1$ comparaciones y una extra para salir del for, la primer asignación es una operación mas, y la suma y asignación por cada iteración, multiplican la variable l. En total tenemos la siguiente función:

$$g(a,b) = (3min(a,b) - 1) + 4 \tag{25}$$

Dentro del for está un if, en el cuál asumiré que siempre es válido y en todas las iteraciones siempre entrará y actualizará la variable r. En éste if tenemos 3 comparaciones y dentro una asignación, por lo tanto se aumenta el numero de operaciones en el for, y queda de la siguiente manera:

$$g(a,b) = (7min(a,b) - 1) + 4 \tag{26}$$

Ahora, para calcular la función complejidad espacial únicamente hay que contar el número de variables que se utilizaron para crear el algoritmo:

$$h(a,b) = 5 \tag{27}$$

6.3. La función complejidad temporal f(n)

$$f(a,b) = (7min(a,b) - 1) + 4 \tag{28}$$

6.4. La función complejidad espacial f(n)

$$f(a,b) = 5 \tag{29}$$

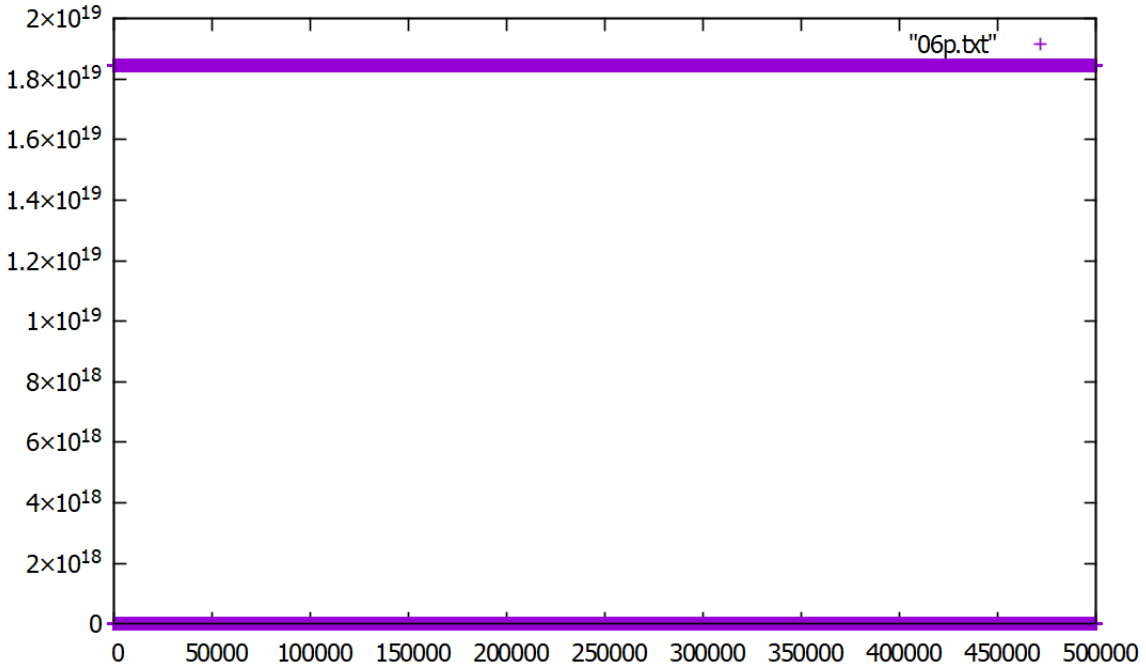
En este caso la función no se puede poner en términos de n, ya que se necesitan dos parámetros a, b, por lo tanto utilice 40 números de los 10 millones que nos dió el profesor para testear el código, 20 para a y 20 para b, los resultados fueron los siguientes:

Cuadro 5: Tabla comparativa del código 1

a	b	Conteo Teórico	Conteo Empírico
856834115	870128690	1702871512	1702871512
654552922	207442309	1452096166	1452096166
1511588122	834068396	1543511479	1543511479
1846873530	966245083	18446744071883332608	18446744071883332608
1799388853	494387258	18446744072875295129	18446744072875295129
984674766	846455593	1630221858	1630221858
178804962	118245790	827720533	827720533
1920624725	839426280	1581016667	1581016667
187130533	1990378944	1309913734	1309913734
525017507	2083630786	18446744073089706872	18446744073089706872
312394492	1380390115	18446744071601345767	18446744071601345767
449643443	1612358838	18446744072562088424	18446744072562088424
1054413206	1158854875	18446744072500509469	18446744072500509469
1328639450	1475318056	710541561	710541561
604218959	1698402220	18446744073644117036	18446744073644117036
220584981	1461053074	1544094870	1544094870
421047262	875137903	18446744072361915157	18446744072361915157
1668495383	1932635384	18446744072504117412	18446744072504117412
1709206300	1367885266	985262273	985262273
751396819	1361111505	964810440	964810440

6.5. Tabla comparativa de la función complejidad

6.6. Gráfica muestral sobre 500,000 puntos



6.6.1. Código de la función complejidad f(a,b)

```
typedef unsigned long long int ull;

int min(int a, int b) {
    return (a < b) ? a : b;
}

ull f(int a, int b) {
    return 1LL * ((7*min(a,b)-1) + 4);
}
```

7. Código 07

7.1. Código implementado por el profesor

```
1 // prof code
```

7.2. Análisis del código

Comencemos con el primer for, es controlado por una variable i que incia en 1 y termina hasta $n-1$, por lo tanto las comparaciones que hará el for serán $n-1$ y una extra para salir del for, también hay que contar la primer asignación de for cuando i se iguala la 1, por ultimo las actualizaciones de dicha variable, es decir una suma y una asignación por cada iteración.

$$(3n-1)+2 \tag{30}$$

Dentro del for, hay un for anidado, el cual es controlado por la variable j, incia en 0 y termina en n-1, por lo tanto, se realizan $n-2$ comparaciones, una operación mas donde se resta a la n, la asignación de j al principio, la última comparación cuando termina el for y las dos operaciones al actualizar la variable j por cada iteración.

$$(3n-2)+3 \tag{31}$$

Asumiremos que e if, siempre se cumple y que va a entrar por cada iteración en el for, comenzamos con las dos consultas al arreglo, una suma y una comparación, dentro del if, se hacen tres asignaciones, cuatro consultas al arreglo y dos sumas. En total se hacen 13 operaciones.

$$(16n-2)+3 \tag{32}$$

Para calcular la complejidad espacial asumiremos que el arreglo es de tamaño n, por lo tanto se cuanta n más 3 variables que utilizaron: $i, j, temp$.

7.3. La función complejidad temporal f(n)

$$f(n) = 48n^2 - 38n + 10 \tag{33}$$

7.4. La función complejidad espacial f(n)

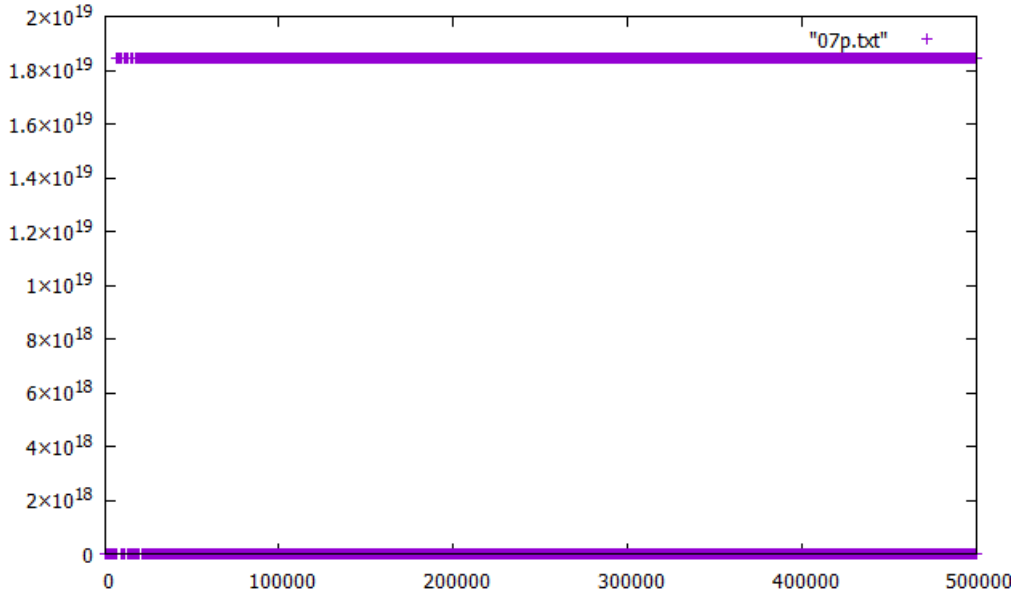
$$f(n) = n + 3 \tag{34}$$

Cuadro 6: Tabla comparativa del código 7

n	Conteo Teórico	Conteo Empírico
-1	96	96
0	10	10
1	20	20
2	126	126
3	328	328
5	1020	1020
15	10240	10240
20	18450	18450
100	476210	476210
409	8013956	8013956
500	11981010	11981010
593	16856628	16856628
1000	47962010	47962010
1471	103808480	103808480
1500	107943010	107943010
2801	376482420	376482420
3000	431886010	431886010
5000	1199810010	1199810010
10000	504652714	504652714
20000	2019370826	2019370826

7.5. Tabla comparativa de la función complejidad

7.6. Gráfica muestral sobre 500,000 puntos



7.6.1. Código de la función complejidad f(n)

```
typedef unsigned long long int integer;

integer f(int n) {
    return (48 * n * n) - (38*n) + 10;
}
```