UNIVERSIDAD
DE GRANADA

Universidad de Granada

Facultad de Ingeniería Informática y telecomunicaciones

Práctica 1: Eficiencia de algoritmos

Doble Grado Ingeniería Informática y Matemáticas

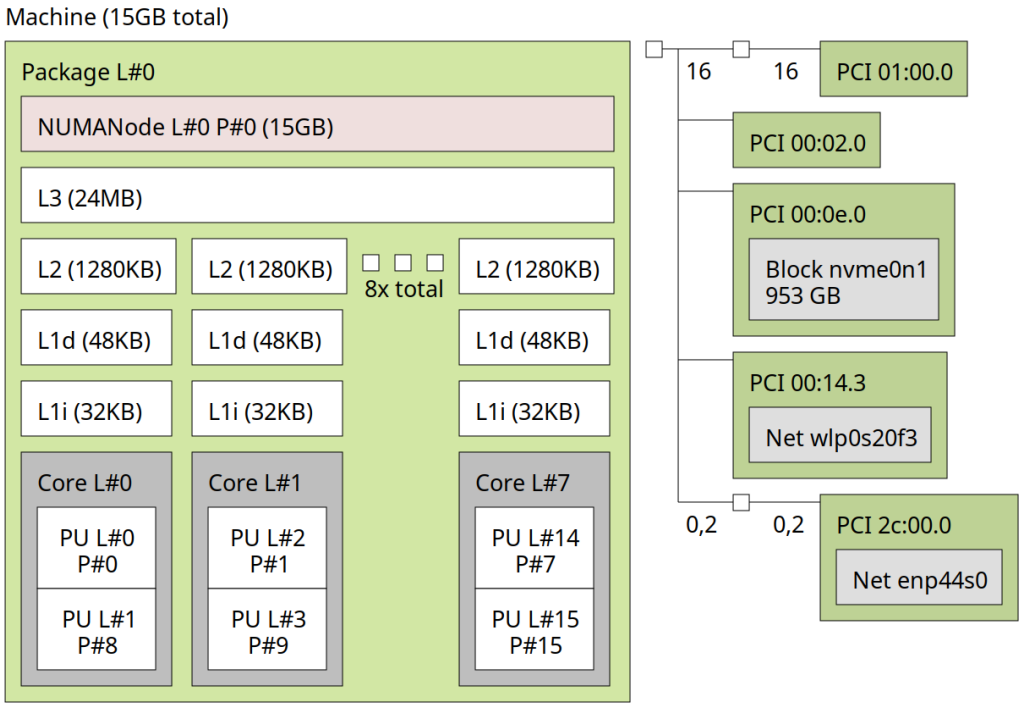
Autores:
Adolfo Martínez Olmedo, Pablo Delgado Galera, Marcos Baena Solar

Marzo 2025

Índice

1. Introducción

Comenzemos estableciendo las características del ordenador que hemos usado para la ejecución de los progrmas y la medición de tiempos de ejecución.



En esta práctica vamos a discutir la eficiencia de los algoritmos desde tres puntos de vista distintos:

1.1. Análisis de la eficiencia teórica

En el análisis de la eficiencia teórica estudiaremos el tiempo de ejecución del algoritmo mediante funciones en notación *Big-O*, que representarán el peor caso posible. En este análisis, no usaremos medidas reales de computación, sino que calcularemos funciones mediante técnicas vistas en Estructuras de Datos y Algorítmica.

1.2. Análisis de la eficiencia empírica

Para el análisis de la eficiencia empírica ejecutaremos los algoritmos implementados en C++ en cada una de nuestras máquinas y mediermos el tiempo de ejecución mediante la clase <chrono>. Cada miembro del equipo ejecutará cada algoritmo 10 veces con todos los tamaños especificados, para luego hacer una media y obtener resultados más fiables.

1.3. Análisis de la eficiencia híbrida

En el análisis de la eficiencia hibrída, tomammos los resultados de los integrantes del grupo y hallamos la constante κ. En la representación de los resultados usaremos la herramienta gnuplot.

Para poder completar esta parte del estudio de la eficiencia usaremos los resultados del análisis teórico, para poder conocer la forma de la función a la que queremos ajustar los datos. Por ejemplo para representar en gnuplot $O(n^3)$:

```
gnuplot> f(x) = a0*x*x+a1*x+a2
```

Ejemplo de $O(n^2)$

Después de esto debemos hacer la regresión usando el método de mínimos, cuyo funcionamiento conoces gracias a la asignatura EDIP:

```
gnuplot> fit f(x) 'result.dat' via a0,a1,a2
```

Uso de gnuplot para la regresión

En este caso result.dat es nuestro fichero de datos. Nos centraremos en *Final set of Parameters*, que nos muestra los coeficientes de la fórmula de regresión junto con la bondad del ajuste realizado.

Finalmente, hacemos el plot de los puntos y la curva de ajuste para ver como de buena es el cálculo de la eficiencia híbrida. Usaremos el siguiente comando:

```
gnuplot> plot 'result.dat', f(x) title 'Curva de ajuste'
```

Representación de la regresión

2. Desarrollo

Una vez que hemos discutido las maneras de estudiar la eficiencia, veamos los problemas que vamos a analizar: La **Ordenación de vectores**, los **Números de Catalan**, y las **Torres de Hanoi.**

2.1. Ordenación de vectores

2.1.1. QuickSort

Eficiencia teórica (caso promedio) A continuación se presenta el código que hemos usado para hacer el análisis:

```
int partition(vector<int> &vec, int low, int high) {
        int pivot = vec[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
                if (vec[j] <= pivot) {
                        i++;
                        swap(vec[i], vec[j]);
                }
        }
        swap(vec[i + 1], vec[high]);
        return i + 1;
}
```

```
14
15 void QuickSort(vector<int> &vec, int low, int high) {
16         if (low < high) {
17                 int pi = partition(vec, low, high);
18                 QuickSort(vec, low, pi - 1);
19                 QuickSort(vec, pi + 1, high);
20         }
21 }
```

Código de QuickSort

Tras analizar el código hemos llegado a la siguiente fórmula de recurrencia:

$$T(n) = 2 \cdot T(n/2) + n$$

Concluimos que en las dos llamadas recursivas de QuickSort se utiliza *pi* que es el resultado de hacer la partición por la mitad del vector. En cuanto al término independiente este se corresponde al bucle de la función auxiliar *partition*, que en el peor de los casos recorre el vector totalmente.

Procedemos a resolver la fórmula de recurrencia utilizando el cambio de variable $n = 2^h$:

$$T(2^h) - 2 \cdot T(2^{h-1}) = 2^h$$

El resultado de la parte homogénea es el siguiente:

$$\begin{aligned} x - 2 &= 0 \\ x &= 2 \end{aligned}$$

El resultado de la parte independiente es:

$$\begin{aligned} p(h) &= 1 \\ b &= 2 \\ d &= 0 \end{aligned}$$

Por tanto la solución es $(x-2)^2$, con multiplicidad 2,y deshaciendo el cambio de variable y expresando la ecuación con coeficientes llegamos finalmente a:

$$c_1 \cdot 2^{\log_2(n)} + c_2 \cdot \log_2(n) \cdot 2^{\log_2(n)}$$

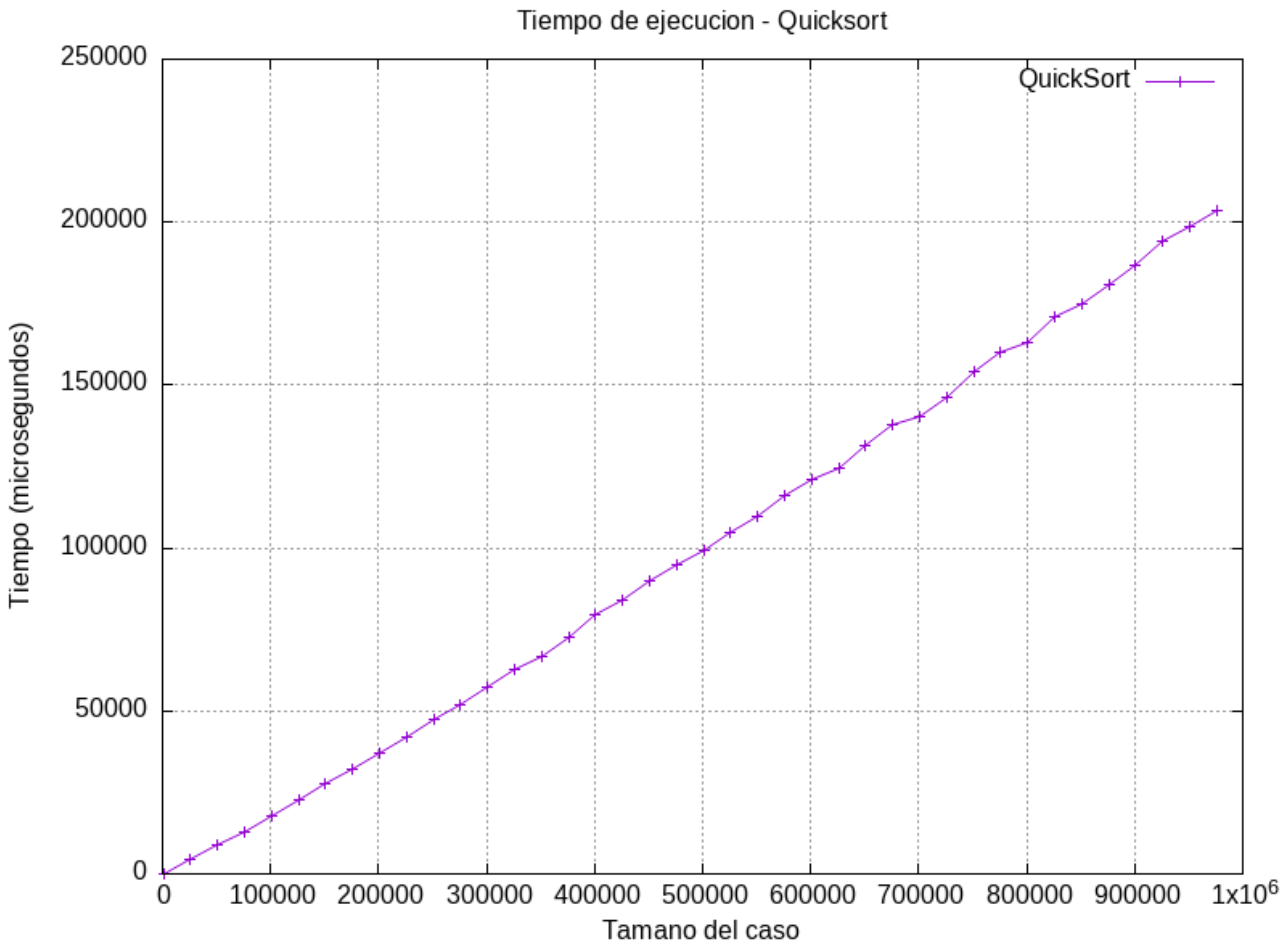
De donde se deduce que QuickSort tiene una eficiencia de $O(n \cdot \log(n))$

Eficiencia práctica: A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *QuickSort* sobre tamaños empezando desde 1.000 hasta 1.000.000 con saltos de 25.000:

Tam.caso	**Tiempo (us)**	K **(nuevo)**	**Tiempo teórico estimado (nuevo)**
1000	135	0,01354635	99,35709574
26000	3923	0,010287898	3801,710723
51000	7972	0,009995628	7951,418511
76000	12281	0,009966386	12285,23441
101000	16285	0,00969909	16739,56621
126000	21136	0,009900572	21283,83619
151000	25810	0,009935205	25899,92812
176000	30765	0,010031505	30575,82905
201000	34579	0,009765365	35302,98061
226000	39644	0,009862605	40074,97275
251000	44586	0,009903006	44886,82354
276000	48868	0,009796113	49734,54906
301000	53481	0,009762833	54614,8904
326000	58561	0,00980834	59525,13206
351000	63962	0,009892341	64462,97638
376000	69415	0,009968177	69426,45388
401000	73753	0,009881282	74413,85763
426000	78161	0,00981132	79423,69396
451000	83203	0,009822066	84454,64485
476000	89583	0,00997845	89505,53863
501000	93698	0,009877337	94575,32688
526000	98931	0,00989659	99663,06596
551000	104380	0,009932909	104767,902
576000	109009	0,009889978	109889,0588
601000	113194	0,009811049	115025,8276
626000	118932	0,009866492	120177,5584
651000	123600	0,009831132	125343,6535
676000	127280	0,009722068	130523,5609
701000	134006	0,009844148	135716,7696
726000	138300	0,009784267	140922,8051
751000	146089	0,009966259	146141,2254
776000	147986	0,009746834	151371,6182
801000	156995	0,009994102	156613,5976
826000	158179	0,00974268	161866,8017
851000	165306	0,009860962	167130,8909
876000	173671	0,010043	172405,5452
901000	176034	0,009876882	177690,4631
926000	180388	0,009828307	182985,3596
951000	186721	0,009886746	188289,9653
976000	189852	0,009776608	193604,0249
Media aritmética de K		**0,009969822**	

Resultados de tiempos y estimaciones de QuickSort (nuevos datos).

Tras graficar los resultados haciendo la curva de ajuste obtenemos:



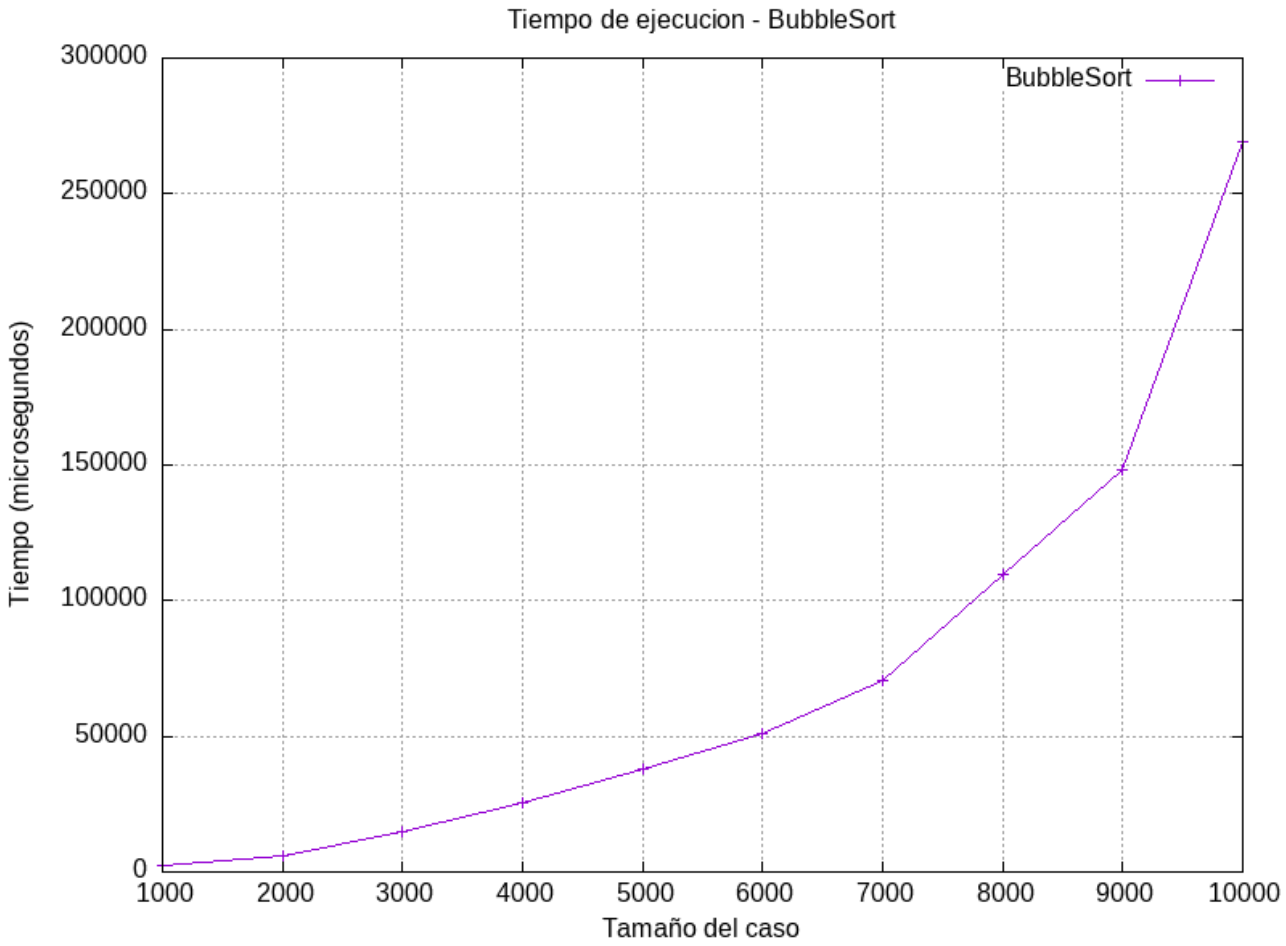
Ajuste por regresión de *QuickSort*.

2.1.2. BubbleSort

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *BubbleSort* sobre tamaños empezando desde 1.000 hasta 10.000 con saltos de 1.000:

Tam.caso	**Tiempo (us)**	K = **Tiempo**$/f(n)$	**Tiempo teórico estimado** $= K \cdot f(n)$
1000	2476	0.002476	1768.2078
2000	6132	0.003066	7020.8312
3000	14902	0.004967	15913.8072
4000	25212	0.00157575	28921.3271
5000	38002	0.0016004	46326.8609
6000	50808	0.001468	63655.4809
7000	70807	0.00173455	86642.1199
8000	109808	0.0013735	116315.2989
9000	148216	0.0016489	143224.2831
10000	269088	0.00182917	176820.7946
Media aritmética de K		**0.001768**	

Resultados de tiempos y estimaciones de BubbleSort.



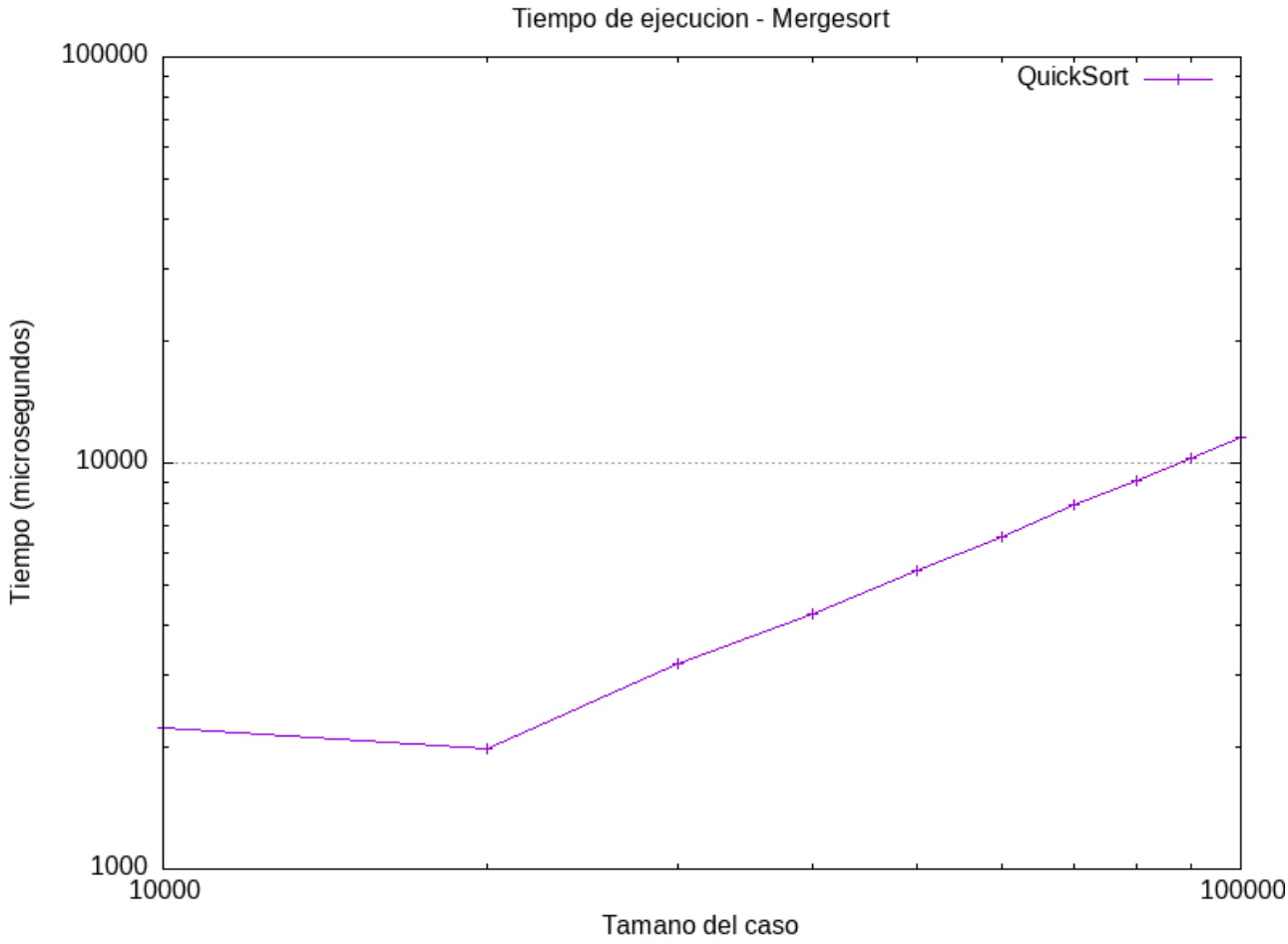
Ajuste por regresión de *BubbleSort*.

2.1.3. MergeSort

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *MergeSort* sobre tamaños empezando desde 10.000 hasta 100.000 con saltos de 1.000:

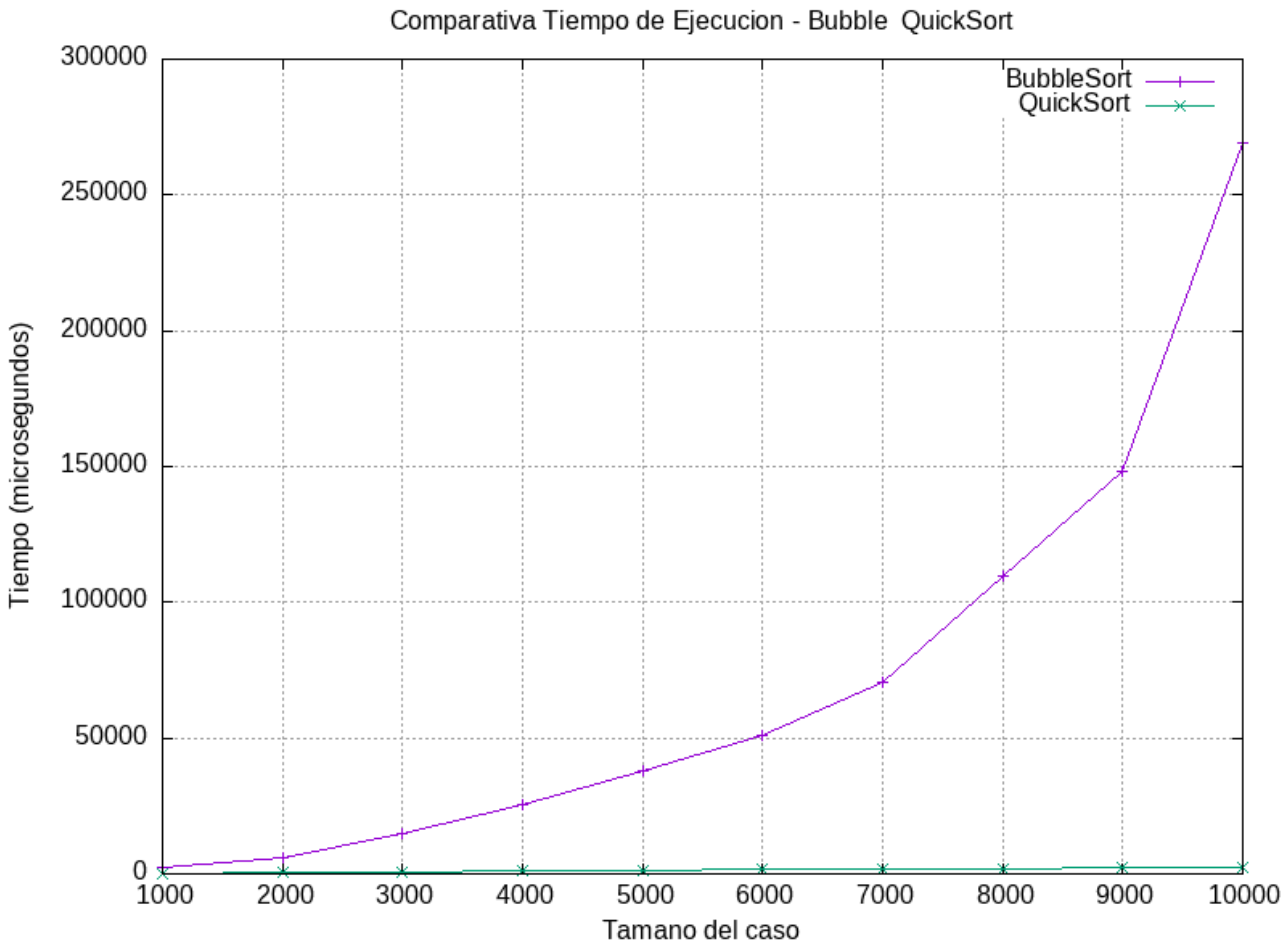
Tam. Caso	**Tiempo (us)**	K = **Tiempo**$/f(n)$	**Tiempo teórico estimado** $= K \cdot f(n)$
10000	2220	0.0555	1054.6881
20000	1972	0.0229247	2268.1228
30000	3198	0.0238099	3541.4752
40000	4262	0.0231527	4853.7383
50000	5419	0.0230646	6194.9352
60000	6580	0.0229517	7559.1895
70000	7864	0.0231869	8942.6188
80000	9009	0.0229677	10342.4622
90000	10265	0.0230218	11756.6574
100000	11546	0.023092	13183.6024
K promedio		**0.0263672**	

Resultados de tiempos y estimaciones de MergeSort.



Ajuste por regresión de *MergeSort*.

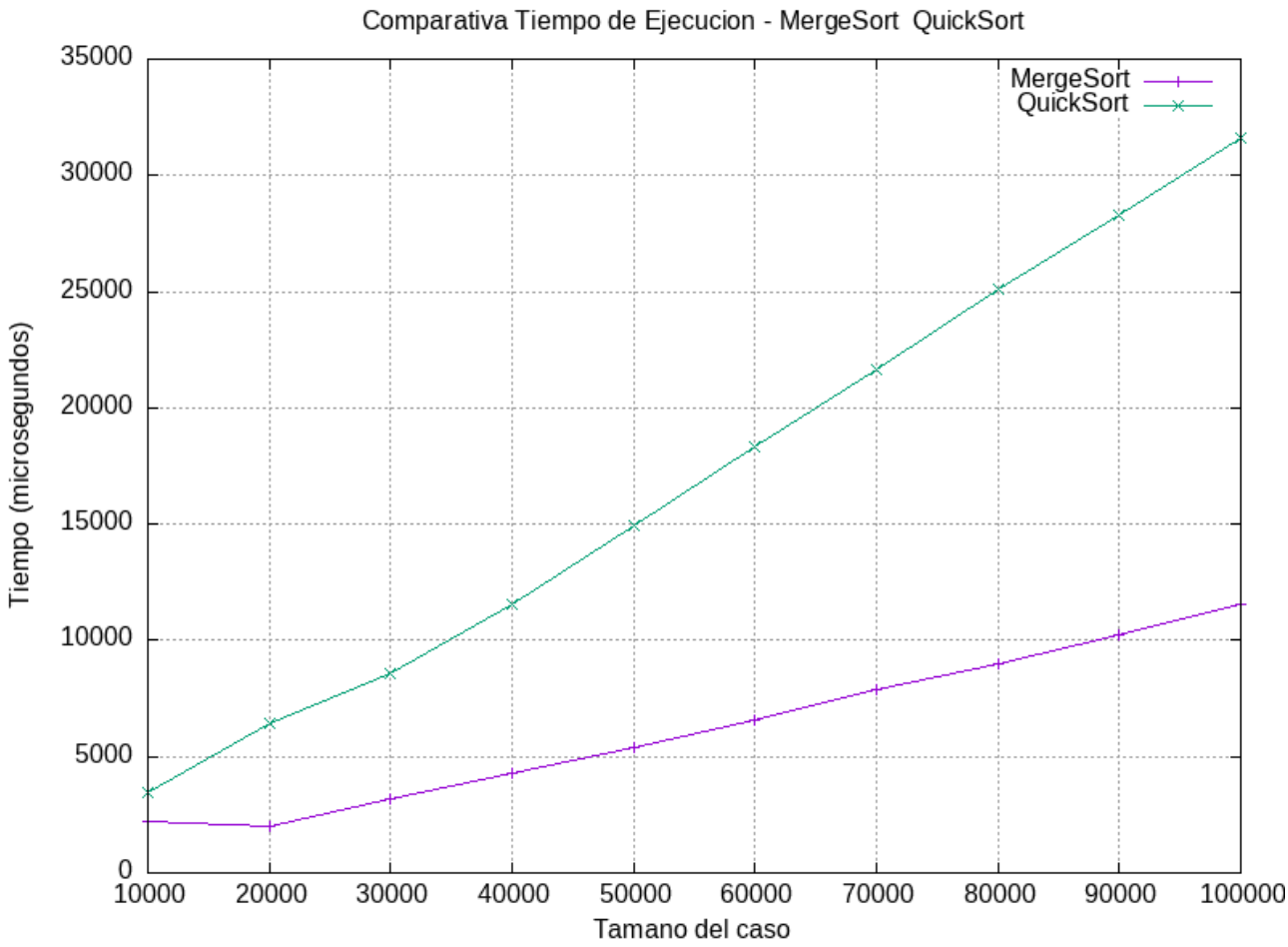
Comparación A continuación se presenta la comparación del algoritmo QuickSort con BubbleSort:



Comparación de QuickSort frente a BubbleSort.

Gracias al estudio de la eficiencia teórica ya aprendimos que el algoritmo QuickSort es más eficiente que BubbleSort y ahora podemos ver esta mejora de eficiencia de manera gráfica, sin embargo observamos que para casos de tamaños muy pequeños, la gráfica de BubbleSort se encuentra por debajo que la de QuickSort, esto es debido a que para dichos casos $O(n^2)$ está por debajo de $O(n \cdot \log(n))$.

Ahora veremos la comparación de QuickSort frente a MergeSort:



Comparación de QuickSort frente a MergeSort.

2.2. Los números de Catalan

2.2.1. Versión recursiva

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1
2  unsigned long int catalan(unsigned int n)
3  {
4  // Base case
5  if (n <= 1)
6          return 1;
7
8  // catalan(n) is sum of
9  // catalan(i)*catalan(n-i-1)
10 unsigned long int res = 0;
11 for (int i = 0; i < n; i++)
12         res += catalan(i) * catalan(n - i - 1);
13
```

```
14 return res;
15 }
```

Código de Catalan recursivo

Para conseguir la eficiencia de este algoritmo, lo primero que tenemos que observar es que

$$T(n) = \sum_{i=0}^{n-1} T(i) \cdot T(n-1-i),$$

sabiendo esto es fácil ver que

$$T(n) = 2 \cdot \sum_{i=0}^{n-1} T(i),$$

ya que se recorre el bucle dos veces: una creciendo y otra decreciendo, variando desde 0 hasta $n-1$.

Para resolver esta recurrencia planteamos una función auxiliar

$$S(n) = \sum_{i=0}^{n} T(i),$$

y observamos que

$$T(n) = S(n) - S(n-1),$$

ya que se anulan todos los términos entre sí, quedando solo $T(n)$. Sustituyendo, tenemos que

$$2 \cdot \sum_{i=0}^{n-1} T(i) = S(n) - S(n-1) \iff 2S(n-1) = S(n) - S(n-1) \iff S(n) - 3S(n-1) = 0.$$

Resolviendo esta simple recurrencia, tenemos que la ecuación homogénea es

$$S(n) = c_0 \cdot 3^n,$$

dando así que

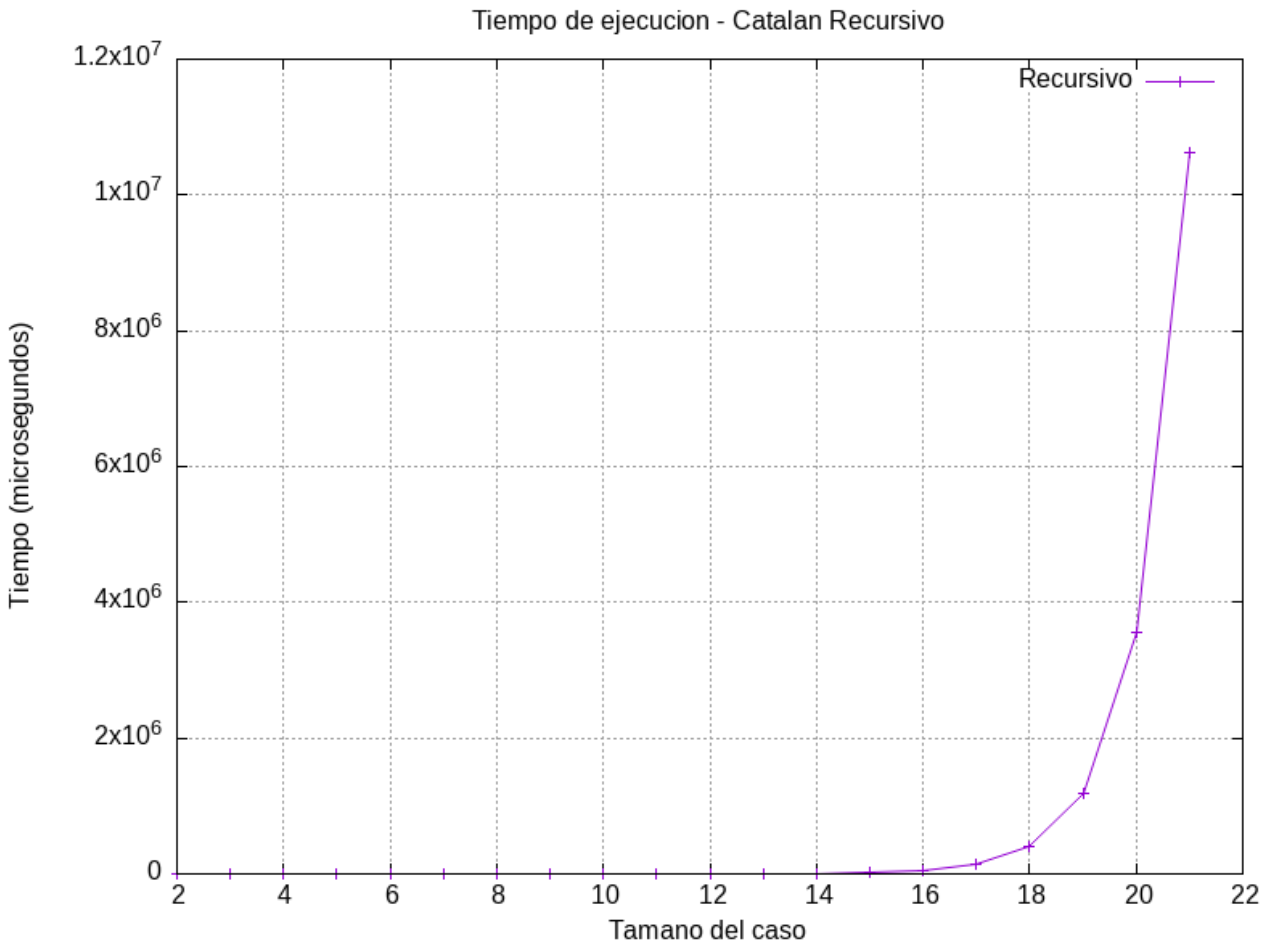
$$T(n) = c_0 \cdot 3^n - c_0 \cdot 3^{n-1} \iff T(n) = 3^{n-1} \cdot c_0 \cdot 2,$$

y, por tanto, la eficiencia del algoritmo es $O(3^n)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *Catalan Recursivo* sobre tamaños empezando con 2 elementos hasta 21 con saltos de 1:

Tam.caso	**Tiempo (us)**	**K = Tiempo/f(n)**	**Tiempo teórico estimado** $= K \cdot f(n)$
2	0	0	0.00911324
3	0	0	0.027339719
4	0	0	0.082019157
5	0	0	0.246057471
6	1	0.001371742	0.738172413
7	3	0.001371742	2.214517238
8	11	0.001676574	6.643551714
9	31	0.001574963	19.93065514
10	96	0.001625768	59.79196543
11	299	0.001687864	179.3758963
12	806	0.001516631	538.1276889
13	1904	0.001194237	1614.383067
14	5147	0.00107611	4843.1492
15	14977	0.001043773	14529.4476
16	44252	0.001027999	43588.3428
17	131750	0.001020209	130765.0284
18	393200	0.001014918	392295.0852
19	1181008	0.001016129	1176885.256
20	3545546	0.001016853	3530655.767
21	10629085	0.001016131	10591967.3
Media aritmética de K		**0.001012582**	

Resultados de tiempos y estimaciones de Catalan Recursivo.



Ajuste por regresión de *Catalan Recursivo*.

2.2.2. Versión iterativa (programación dinámica)

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1
2  unsigned long int catalanDP(unsigned int n)
3  {
4  // Table to store results of subproblems
5  unsigned long int catalan[n + 1];
6
7  // Initialize first two values in table
8  catalan[0] = catalan[1] = 1;
9
10 // Fill entries in catalan[] using recursive formula
11 for (int i = 2; i <= n; i++) {
12         catalan[i] = 0;
13         for (int j = 0; j < i; j++)
14                 catalan[i] += catalan[j] * catalan[i - j - 1];
15 }
```

```
16
17 // Return last entry
18 return catalan[n];
19 }
```

Código de Catalan iterativo (programación dinámica)

Tras analizar el código, vemos que la eficiencia depende de los dos bucles anidados, donde el segundo depende del primero. Veamos la eficiencia, sabiendo que n es el número de Catalan a calcular:

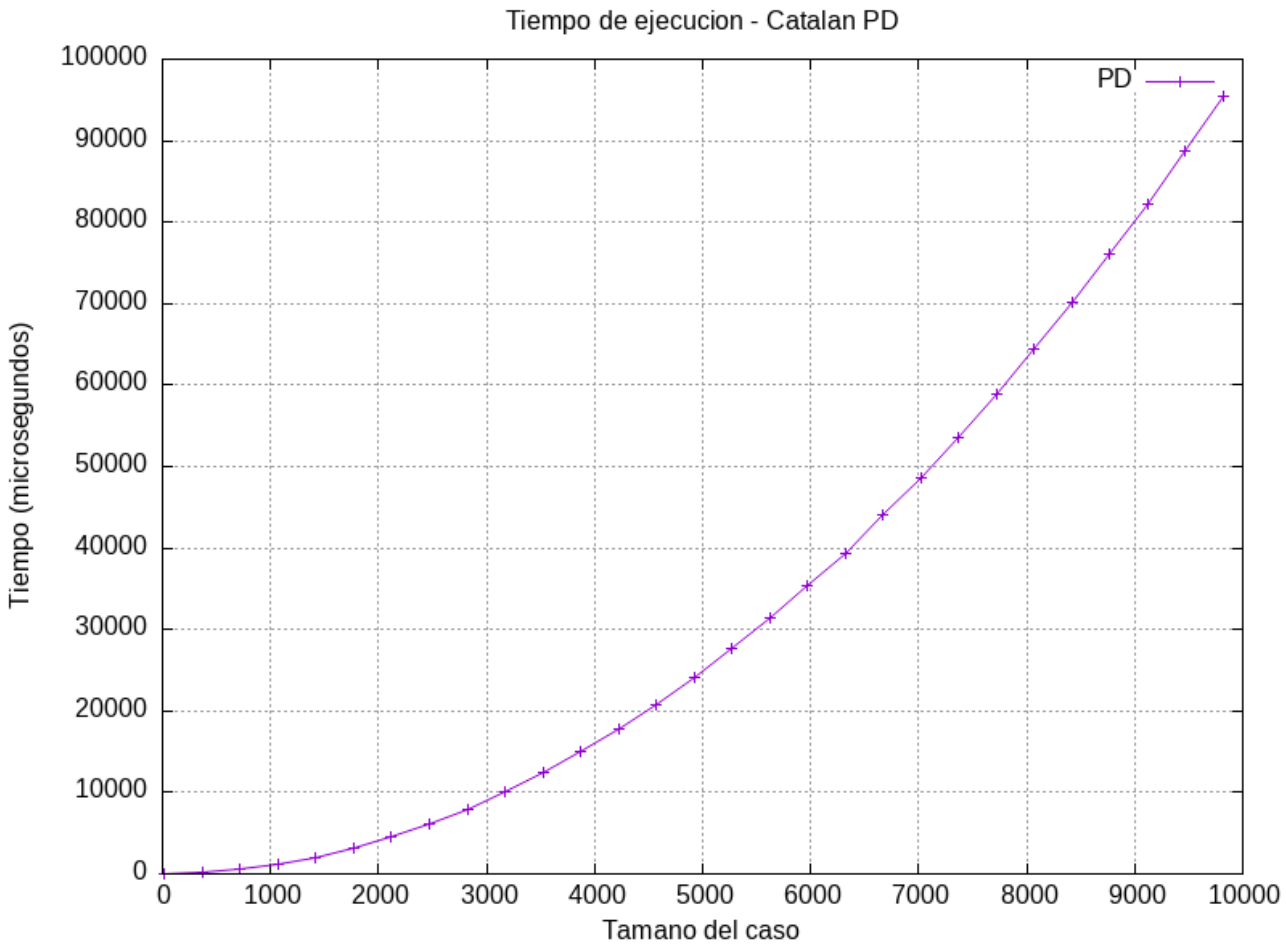
$$\sum_{i=2}^{n}\sum_{j=0}^{i-1} 1 = \sum_{i=2}^{n} i = \frac{n(n+1)}{2}.$$

Dentro del segundo bucle `for`, usamos un 1, ya que todas las operaciones dentro del segundo bucle son $O(1)$. Por tanto, la eficiencia de este algoritmo es $O(n^2)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *Catalan Iterativo PD* sobre tamaños empezando desde 20 hasta 10.000 con saltos de 350:

Tam.caso	**Tiempo (us)**	**K = Tiempo/f(n)**	**Tiempo teórico estimado = K × f(n)**
20	0	0	0.095902338
370	1	0.002702703	1.774193253
720	3	0.004166667	3.452484169
1070	5	0.004672897	5.130775084
1420	9	0.006338028	6.809065999
1770	9	0.005084746	8.487356915
2120	11	0.005188679	10.16564783
2470	12	0.0048583	11.84393875
2820	14	0.004964539	13.52222966
3170	16	0.005047319	15.20052058
3520	17	0.004829545	16.87881149
3870	20	0.005167959	18.55710241
4220	22	0.00521327	20.23539332
4570	23	0.005032823	21.91368424
4920	26	0.005284553	23.59197515
5270	27	0.00512334	25.27026607
5620	29	0.005160142	26.94855698
5970	31	0.00519263	28.6268479
6320	33	0.005221519	30.30513881
6670	34	0.005097451	31.98342973
7020	36	0.005128205	33.66172065
7370	39	0.005291723	35.34001156
7720	40	0.005181347	37.01830248
8070	42	0.005204461	38.69659339
8420	44	0.005225653	40.37488431
8770	45	0.005131129	42.05317522
9120	47	0.005153509	43.73146614
9470	39	0.004118268	45.40975705
9820	42	0.004276986	47.08804797
Media artimética de K		**0.004795117**	

Resultados de tiempos y estimaciones de Catalan iterativo PD.



Ajuste por regresión de *Catalan Iterativo (programación dinámica)*.

2.2.3. Versión iterativa directa usado el coeficiente binomial

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1
2  unsigned long int binomialCoeff(unsigned int n,
3                                                  unsigned int k)
4  {
5  unsigned long int res = 1;
6
7  // Since C(n, k) = C(n, n-k)
8  if (k > n - k)
9          k = n - k;
10
11 // Calculate value of [n*(n-1)*---*(n-k+1)] /
12 // [k*(k-1)*---*1]
13 for (int i = 0; i < k; ++i) {
14         res *= (n - i);
15         res /= (i + 1);
```

```
16 }
17
18 return res;
19 }
20
21 // A Binomial coefficient based function to find nth catalan
22 // number in O(n) time
23 unsigned long int catalan(unsigned int n)
24 {
25     // Calculate value of 2nCn
26     unsigned long int c = binomialCoeff(2 * n, n);
27
28     // return 2nCn/(n+1)
29     return c / (n + 1);
30 }
```

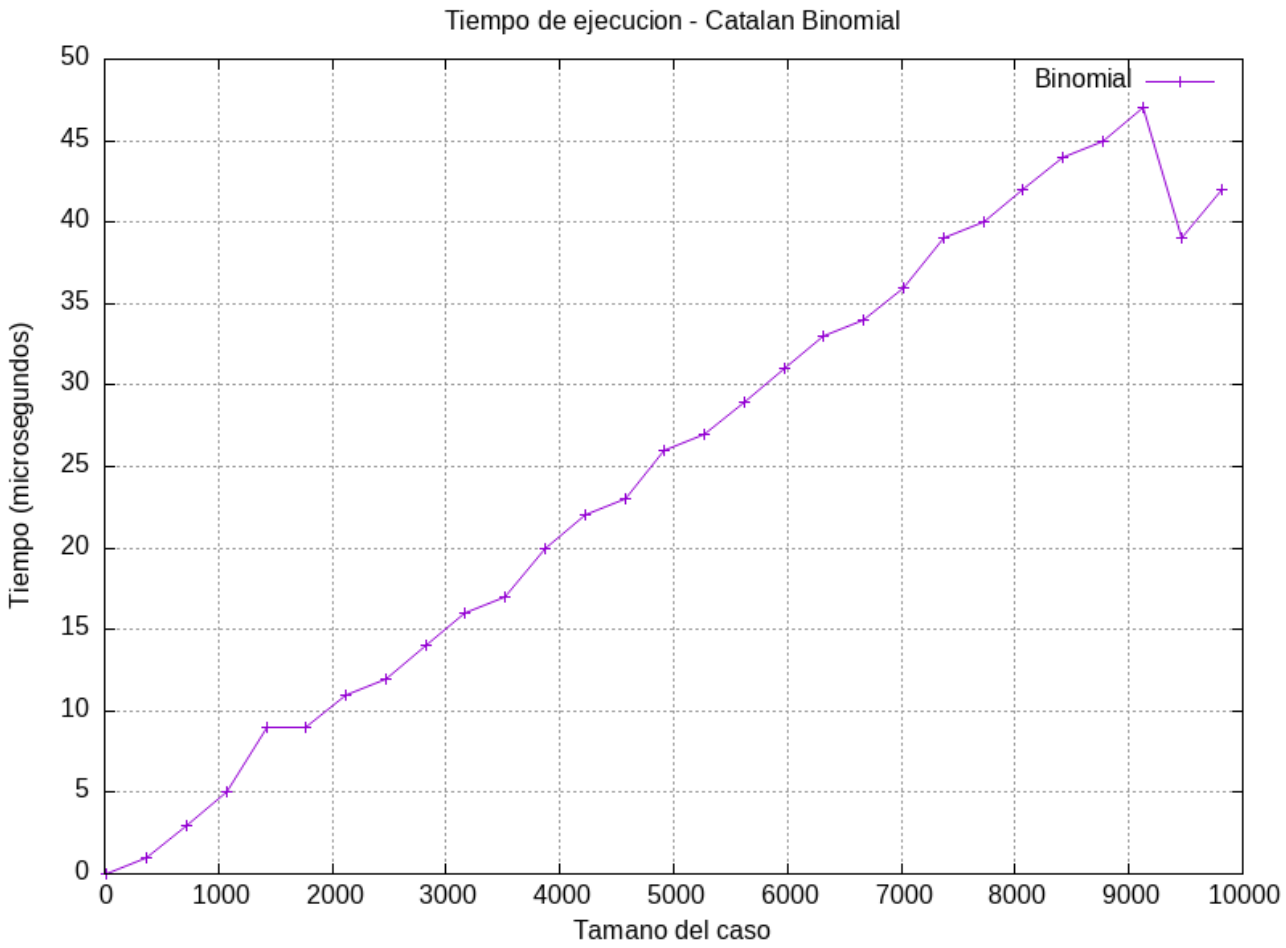
Código de Catalan iterativo binomial

Al ver el código, vemos claramente que la eficiencia de este algoritmo depende de la función *binomialCoeff*, la cual depende únicamente del bucle `for`. Este bucle recorre todos los valores desde 0 hasta $k - 1$. Al llamar a esta función se le pasan dos valores: n (que siempre se llama con $2n$) y k (que se llama con n). Aclaramos que, en este caso, n siempre será mayor que k; por tanto, no importará la condición puesta. Por todo lo dicho, vemos que el bucle nos determina una eficiencia total de $O(n)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *Catalan iterativo binomial* sobre tamaños empezando con 20 hasta 10.000 con saltos de 350:

Tam.caso	**Tiempo (us)**	**K = Tiempo/f(n)**	**Tiempo teórico estimado = K × f(n)**
20	0	0	0.393360124
370	181	0.001322133	134.6275026
720	651	0.001255787	509.7947212
1070	1169	0.00102105	1125.895016
1420	2052	0.001017655	1982.928387
1770	3174	0.001013119	3080.894834
2120	4551	0.001012593	4419.794358
2470	6161	0.001009851	5999.626957
2820	8003	0.001006363	7820.392633
3170	10088	0.001003891	9882.091385
3520	12409	0.001001501	12184.72321
3870	15013	0.00100241	14728.28812
4220	17826	0.001000988	17512.7861
4570	20844	0.000998042	20538.21716
4920	24140	0.000997257	23804.58129
5270	27661	0.000995971	27311.8785
5620	31412	0.000994542	31060.10878
5970	35415	0.000993662	35049.27214
6320	39394	0.00098627	39279.36858
6670	43997	0.000988943	43750.3981
7020	48646	0.000987127	48462.36069
7370	53611	0.000987004	53415.25635
7720	58827	0.000987057	58609.0851
8070	64390	0.000988716	64043.84691
8420	70220	0.000990459	69719.54181
8770	76059	0.000988898	75636.16978
9120	82239	0.000988754	81793.73083
9470	88717	0.000989252	88192.22495
9820	95402	0.000989315	94831.65215
Media artimética de K		**0.0009834**	

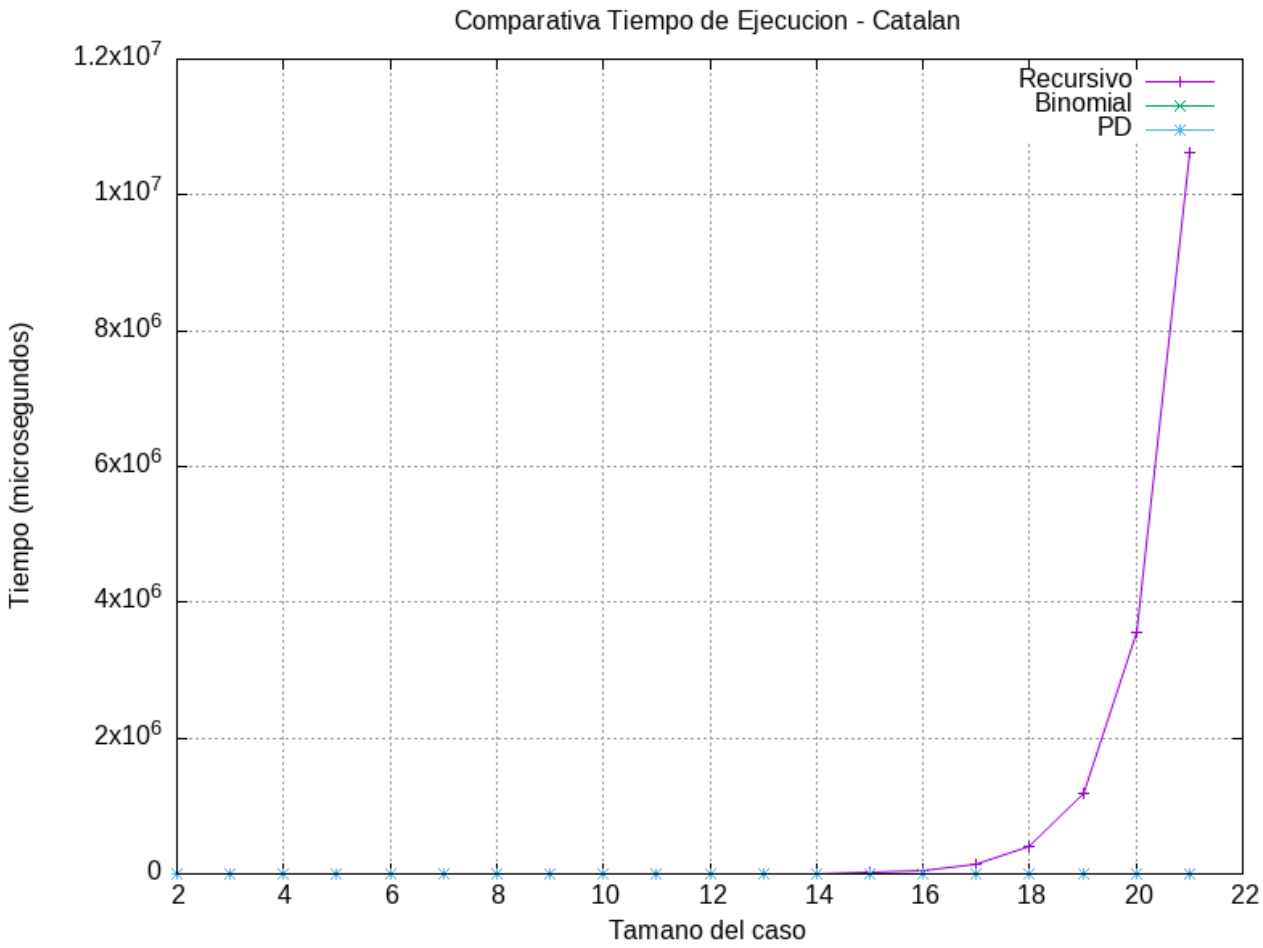
Resultados de tiempos y estimaciones de Catalan Iterativo Binomial



Ajuste por regresión de *Catalan Iterativo usando el Coeficiente Binomial.*

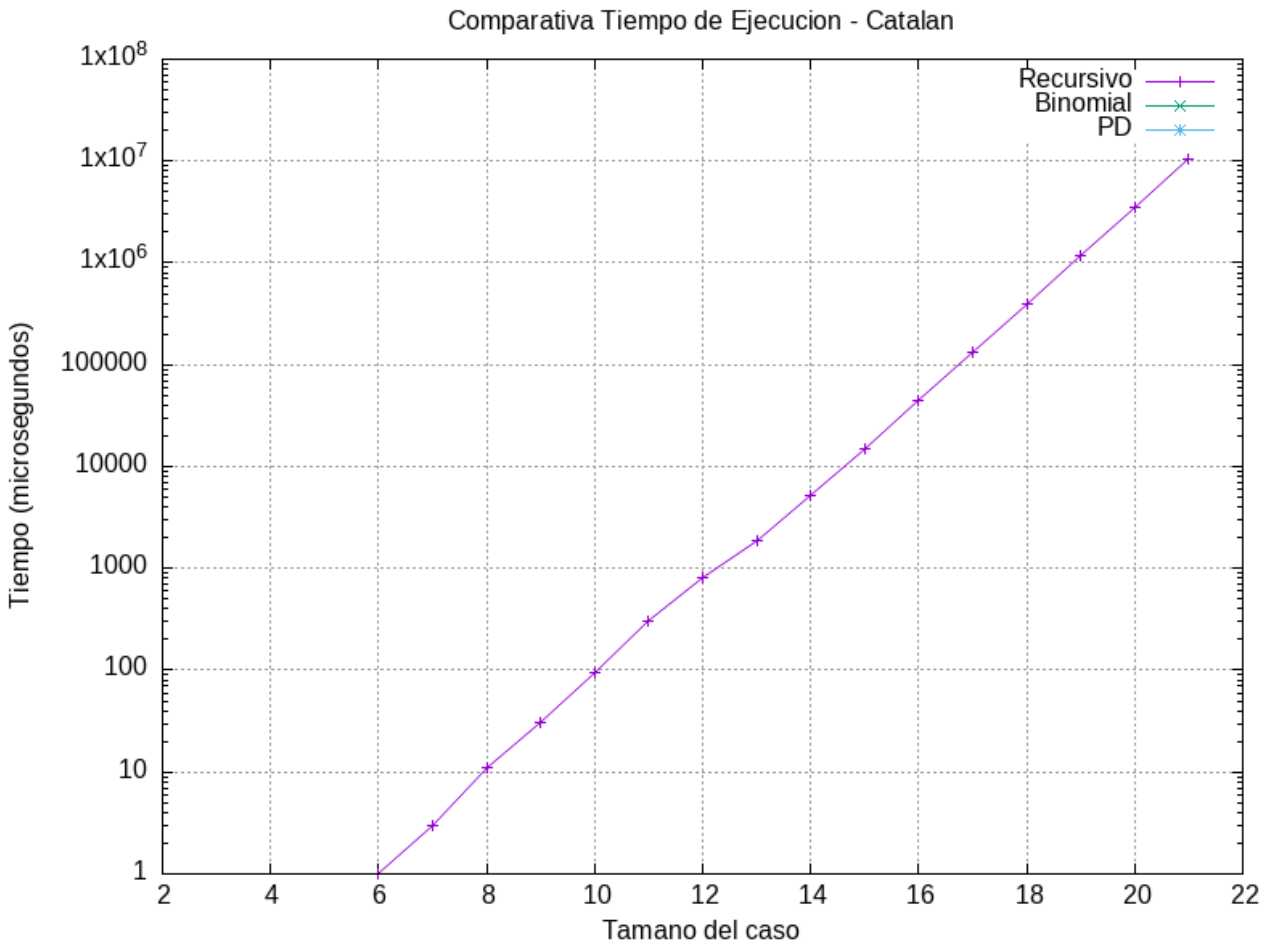
Como hemos visto de manera teórica y empírica, el algoritmo *Catalan Recursivo* es muchísimo menos eficiente que los otros dos. Tanto es así, que obtener un resultado con n >30 usándolo es practicamente imposible usando nuestros ordenadores. Al ser tanto más eficientes los otros dos, para casos muy pequeños, de entre 0 y 25, parecen casi instanáneos.

Comparación A continuación se presenta la comparación de los tres algoritmos de Catalan:



Comparación de los algoritmos de Catalan.

También la comparativa en escala logarítmica:



Comparación de los algoritmos de Catalan en escala logarítmica.

2.3. Las Torres de Hanoi

2.3.1. Versión recursiva

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1  void Hanoi(int n, char from_rod, char to_rod,
2             char aux_rod)
3  {
4      if (n == 0) {
5          return;
6      }
7      Hanoi(n - 1, from_rod, aux_rod, to_rod);
8      cout << "Move disk " << n << " from rod " << from_rod
9           << " to rod " << to_rod << endl;
10     Hanoi(n - 1, aux_rod, to_rod, from_rod);
11 }
```

Código de Hanoi recursivo

Tras analizar el código hemos llegado a la siguiente fórmula de recurrencia:

$$H(n) = \begin{cases} 0, & \text{si } n = 0, \\ 2 \cdot H(n-1), & \text{si } n > 0 \end{cases}$$

Concluimos que en las dos llamadas recursivas de *Hanoi* se utiliza *n - 1*, de donde sale $2 \cdot H(n-1)$, y el caso base 0 ocurre cuando hay 0 discos.

Procedemos a resolver la ecuación:

$$\begin{aligned} x - 2 &= 0 \\ x &= 2 \end{aligned}$$

Por tanto la solución es $(x-2)$, con multiplicidad 1, y expresando la ecuación con coeficientes llegamos finalmente a:

$$c_1 \cdot 2^n$$

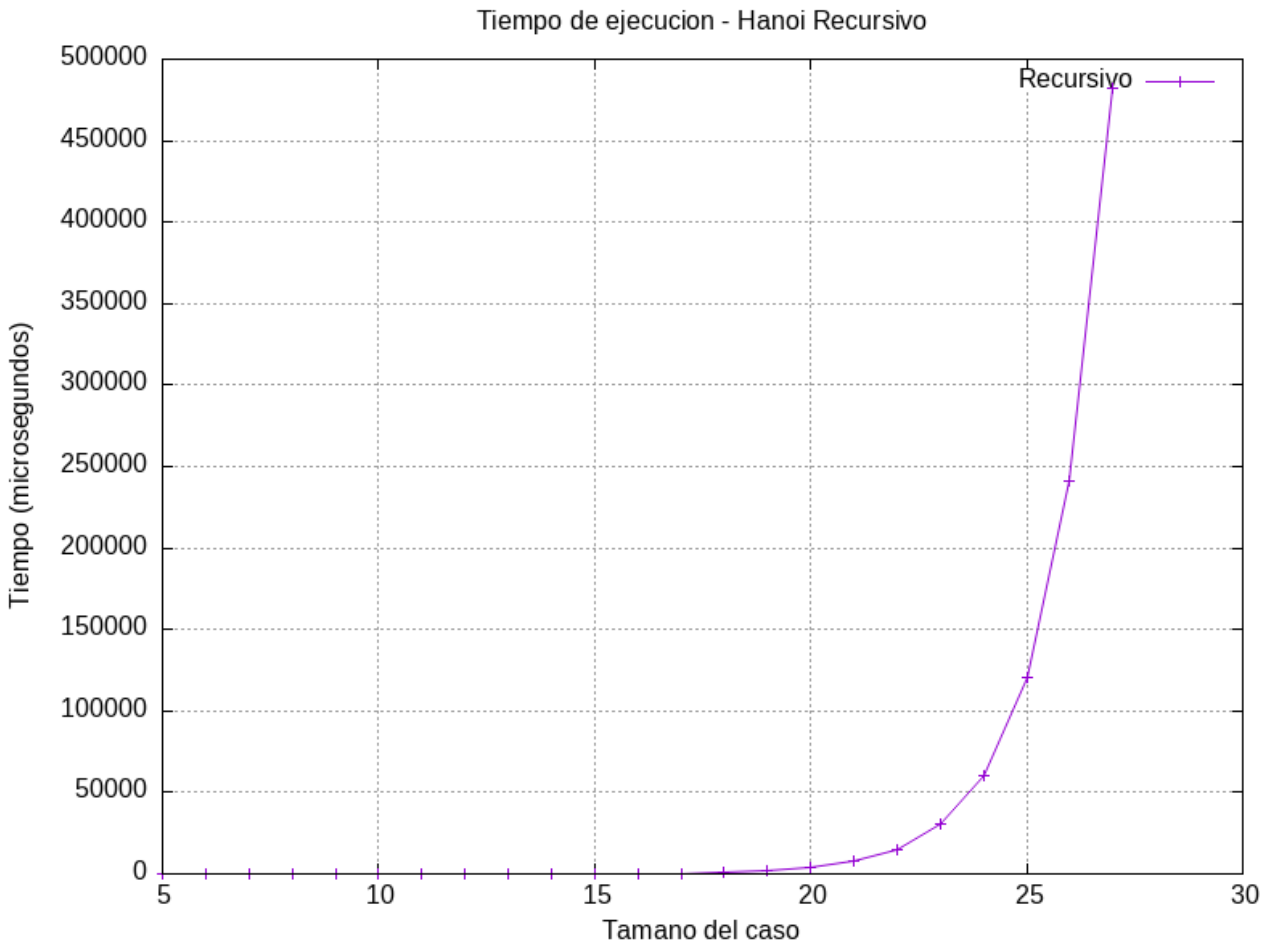
De donde se deduce que Hanoi tiene una eficiencia de $O(2^n)$

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *Hanoi Recursivo* sobre tamaños empezando con 5 elementos hasta 27 con saltos de 1:

Tam.caso	**Tiempo (us)**	$K =$ **Tiempo**$/f(n)$	**Tiempo teórico estimado** $= K \cdot f(n)$
5	0	0	0.093463214
6	0	0	0.186926427
7	0	0	0.373852854
8	0	0	0.747705708
9	1	0.001953125	1.495411417
10	3	0.002929688	2.990822834
11	7	0.003417969	5.981645667
12	15	0.003662109	11.96329133
13	32	0.00390625	23.92658267
14	61	0.003723145	47.85316534
15	123	0.003753662	95.70633067
16	244	0.003723145	191.4126613
17	485	0.003700256	382.8253227
18	974	0.003715515	765.6506454
19	1937	0.003694534	1531.301291
20	3867	0.003687859	3062.602582
21	7632	0.003639221	6125.205163
22	15199	0.003623724	12250.41033
23	30871	0.00368011	24500.82065
24	60305	0.003594458	49001.6413
25	120463	0.003590077	98003.28261
26	240884	0.003589451	196006.5652
27	482162	0.003592387	392013.1304
Media aritmética de K		**0.002920725**	

Resultados de tiempos y estimaciones de Hanoi Recursivo.

La gráfica resultante es:



Ajuste por regresión de *Hanoi Recursivo*.

2.3.2. Versión iterativa usando una pila

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1         void moveDisk(int a, int b)
2         {
3                 if (!stacks[a].empty() && (stacks[b].empty() || stacks[
                     a].top() < stacks[b].top()))
4                 {
5                         stacks[b].push(stacks[a].top());
6                         stacks[a].pop();
7                 }
8         }
9
10        void Hanoi(int n)
11        {
12
13                int src = 0, aux = 1, dest = 2;
14                for (int i = n; i > 0; i--)
```

```
15                          stacks[src].push(i);
16
17                  int totalMoves = (1 << n) - 1;
18                  if (n % 2 == 0)
19                          swap(aux, dest);
20
21                  for (int i = 1; i <= totalMoves; i++)
22                  {
23                          if (i % 3 == 0)
24                                  moveDisk(aux, dest);
25                          else if (i % 3 == 1)
26                                  moveDisk(src, dest);
27                          else
28                                  moveDisk(src, aux);
29                  }
30          }
```

Código de Hanoi Iterativo con pila

Tras analizar el código vemos que:

El primer bucle claramente tiene una eficiencia de $O(\mathrm{n})$ ya que vemos que comienza con $i = n$ (el número de discos) y va decrementando hasta llegar a 0.

El segundo bucle recorre desde $i = 1$ hasta llegar a la variable totalMoves que se obtiene con la función $(1 \ll n) - 1$ que equivale a $2^n - 1$ por tanto el segundo bucle tiene una eficiencia de $O(2^n)$.

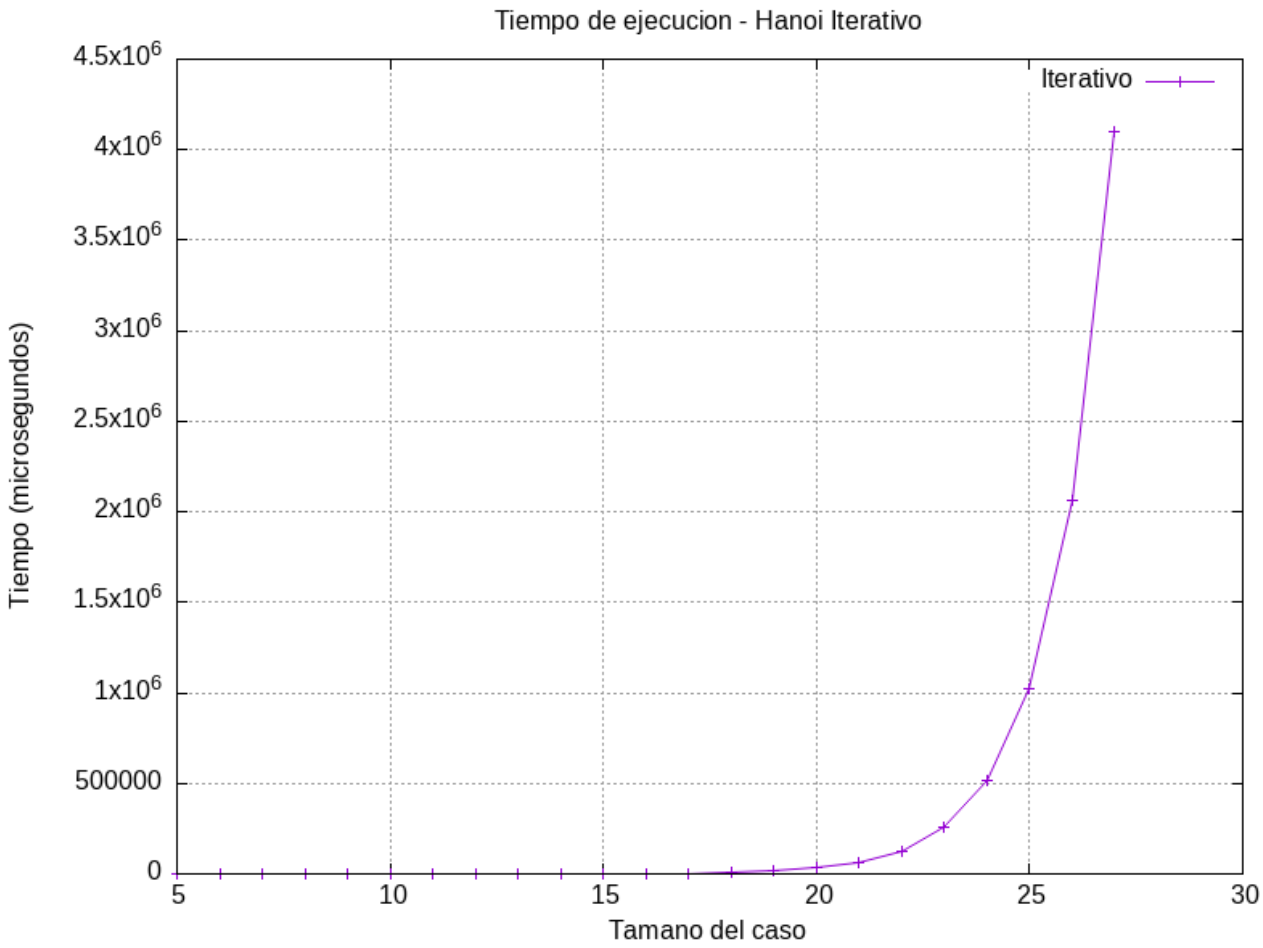
Por tanto, por la regla del máximo, al no estar los bucles anidados obtenemos que la eficiencia total de la función es $O(2^n)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *Hanoi Iterativo Con Pila* sobre tamaños empezando con 5 elementos hasta 27 con saltos de 1:

Tam.caso	**Tiempo (us)**	**K = Tiempo/f(n)**	**Tiempo teórico estimado = K × f(n)**
5	1	0.03125	0.967404925
6	2	0.03125	1.934809851
7	3	0.0234375	3.869619701
8	7	0.02734375	7.739239402
9	16	0.03125	15.4784788
10	30	0.029296875	30.95695761
11	63	0.030761719	61.91391522
12	124	0.030273438	123.8278304
13	249	0.030395508	247.6556609
14	499	0.030456543	495.3113218
15	999	0.030487061	990.6226435
16	2044	0.031188965	1981.245287
17	4107	0.031333923	3962.490574
18	8119	0.030971527	7924.981148
19	16236	0.030967712	15849.9623
20	32090	0.030603409	31699.92459
21	64218	0.030621529	63399.84918
22	127909	0.030495882	126799.6984
23	256123	0.030532241	253599.3967
24	513154	0.030586362	507198.7935
25	1026528	0.030592918	1014397.587
26	2059635	0.030690953	2028795.174
27	4098268	0.030534476	4057590.348
Media aritmética de K		**0.030231404**	

Resultados de tiempos y estimaciones teóricas del Hanoi Iterativo con Pila.

La gráfica resultante es:



Ajuste por regresión de *Hanoi Iterativo Con Pila*.

2.3.3. Versión iterativa sin usar la pila

Eficiencia teórica A continuación se presenta el código que hemos usado para hacer el análisis:

```
1  void moveDisksBetweenTwoPoles(struct Stack *src, struct Stack *
       dest, char s, char d) {
2          int pole1TopDisk = pop(src);
3          int pole2TopDisk = pop(dest);
4
5          if (pole1TopDisk == INT_MIN) {
6                  push(src, pole2TopDisk);
7          } else if (pole2TopDisk == INT_MIN) {
8                  push(dest, pole1TopDisk);
9          } else if (pole1TopDisk > pole2TopDisk) {
10                 push(src, pole1TopDisk);
11                 push(src, pole2TopDisk);
12         } else {
13                 push(dest, pole2TopDisk);
14                 push(dest, pole1TopDisk);
```

```
15                  }
16          }
17
18          void tohIterative(int num_of_disks, struct Stack *src, struct
               Stack *aux, struct Stack *dest) {
19                  char s = 'S', d = 'D', a = 'A';
20                  if (num_of_disks % 2 == 0) swap(d, a);
21
22                  int total_num_of_moves = pow(2, num_of_disks) - 1;
23                  for (int i = num_of_disks; i >= 1; i--) push(src, i);
24
25                  for (int i = 1; i <= total_num_of_moves; i++) {
26                          if (i % 3 == 1) moveDisksBetweenTwoPoles(src,
                               dest, s, d);
27                          else if (i % 3 == 2) moveDisksBetweenTwoPoles(
                               src, aux, s, a);
28                          else moveDisksBetweenTwoPoles(aux, dest, a, d);
29                  }
30          }
```

Código de Hanoi Iterativo sin pila

Tras analizar el código vemos que:

El primer bucle claramente tiene una eficiencia de O(n) ya que vemos que comienza con $i = n$ (el número de discos) y va decrementando hasta llegar a 1.

El segundo bucle recorre desde $i = 1$ hasta llegar a la variable total_num_of_moves que se obtiene con la función pow(2, num_of_disks) - 1 que equivale a $2^n - 1$ por tanto el segundo bucle tiene una eficiencia de $O(2^n)$.

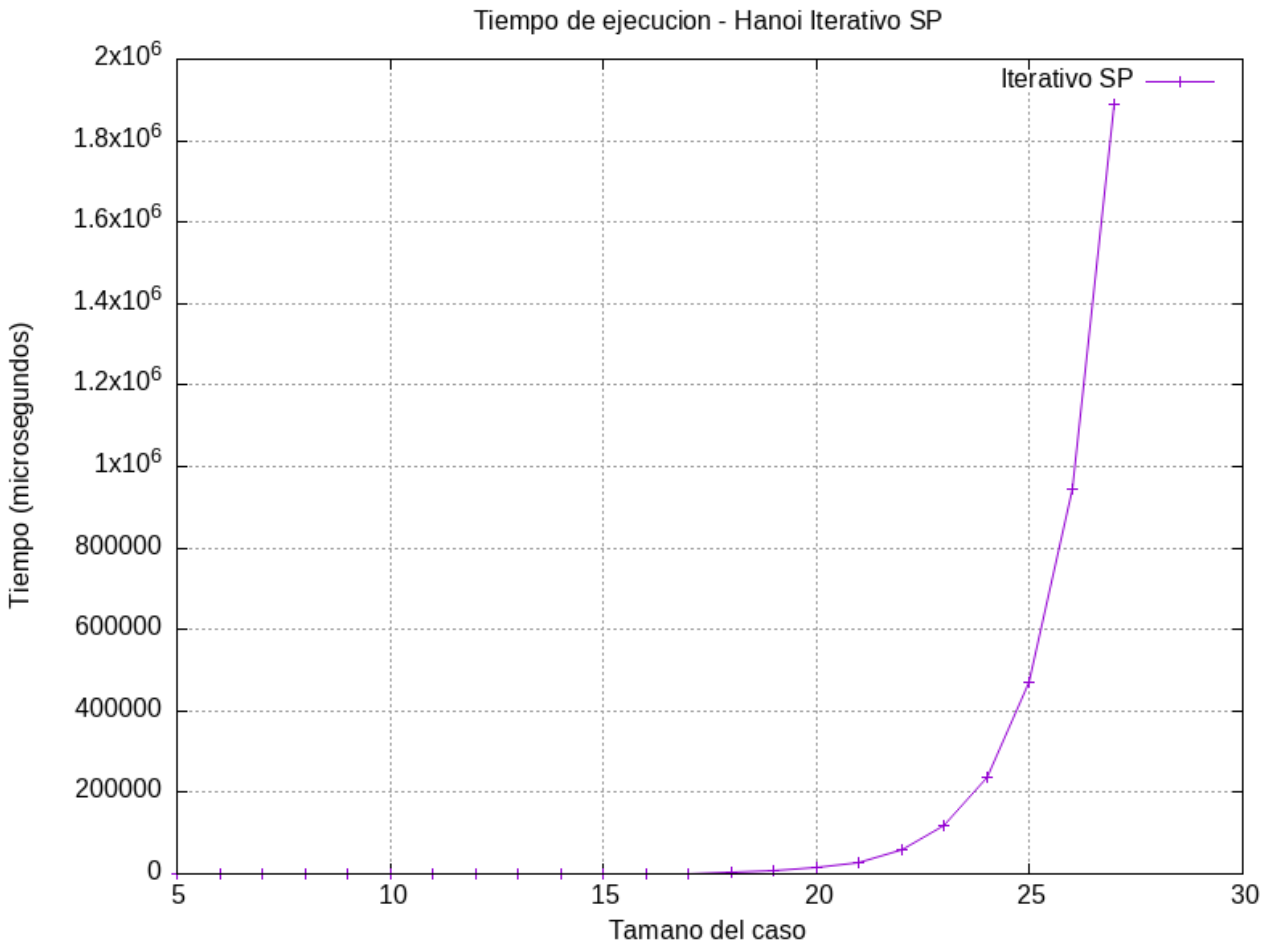
Por tanto, por la regla del máximo, al no estar los bucles anidados obtenemos que la eficiencia total de la función es $O(2^n)$.

Eficiencia práctica A continuación mostramos las gráficas resultantes de ejecutar el algoritmo *Hanoi Iterativo sin pila* sobre tamaños empezando con 5 elementos hasta 27 con saltos de 1:

Tam.caso	**Tiempo (us)**	**K = Tiempo/f(n)**	**Tiempo teórico estimado = K × f(n)**
5	12	0.375	0.952186895
6	1	0.015625	1.904373791
7	1	0.0078125	3.808747582
8	3	0.01171875	7.617495164
9	7	0.013671875	15.23499033
10	14	0.013671875	30.46998065
11	29	0.014160156	60.93996131
12	65	0.015869141	121.8799226
13	132	0.016113281	243.7598452
14	257	0.015686035	487.5196905
15	496	0.015136719	975.0393809
16	924	0.014099121	1950.078762
17	1906	0.014541626	3900.157524
18	3906	0.014900208	7800.315048
19	7271	0.013868332	15600.63010
20	14782	0.014097214	31201.26019
21	29515	0.014073849	62402.52038
22	58605	0.013972521	124805.0408
23	118189	0.014089227	249610.0815
24	236037	0.014068902	499220.1630
25	471250	0.014044344	998440.3261
26	944599	0.014075622	1996880.652
27	1890864	0.014088035	3993761.304
Media aritmética de K		**0.02975584**	

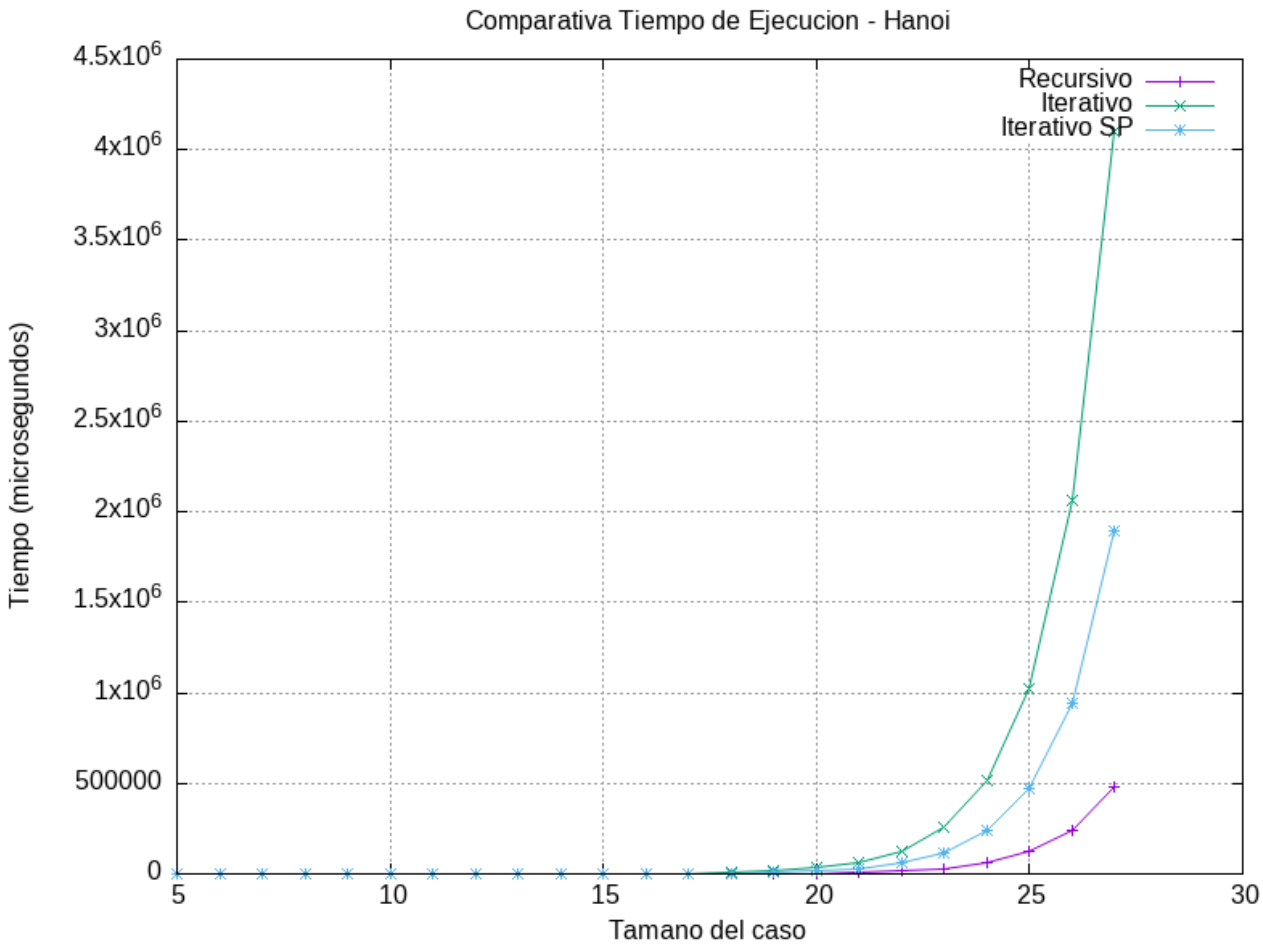
Resultados de tiempos y estimaciones teóricas de Hanoi Iterativo sin Pila.

La gráfica resultante es:



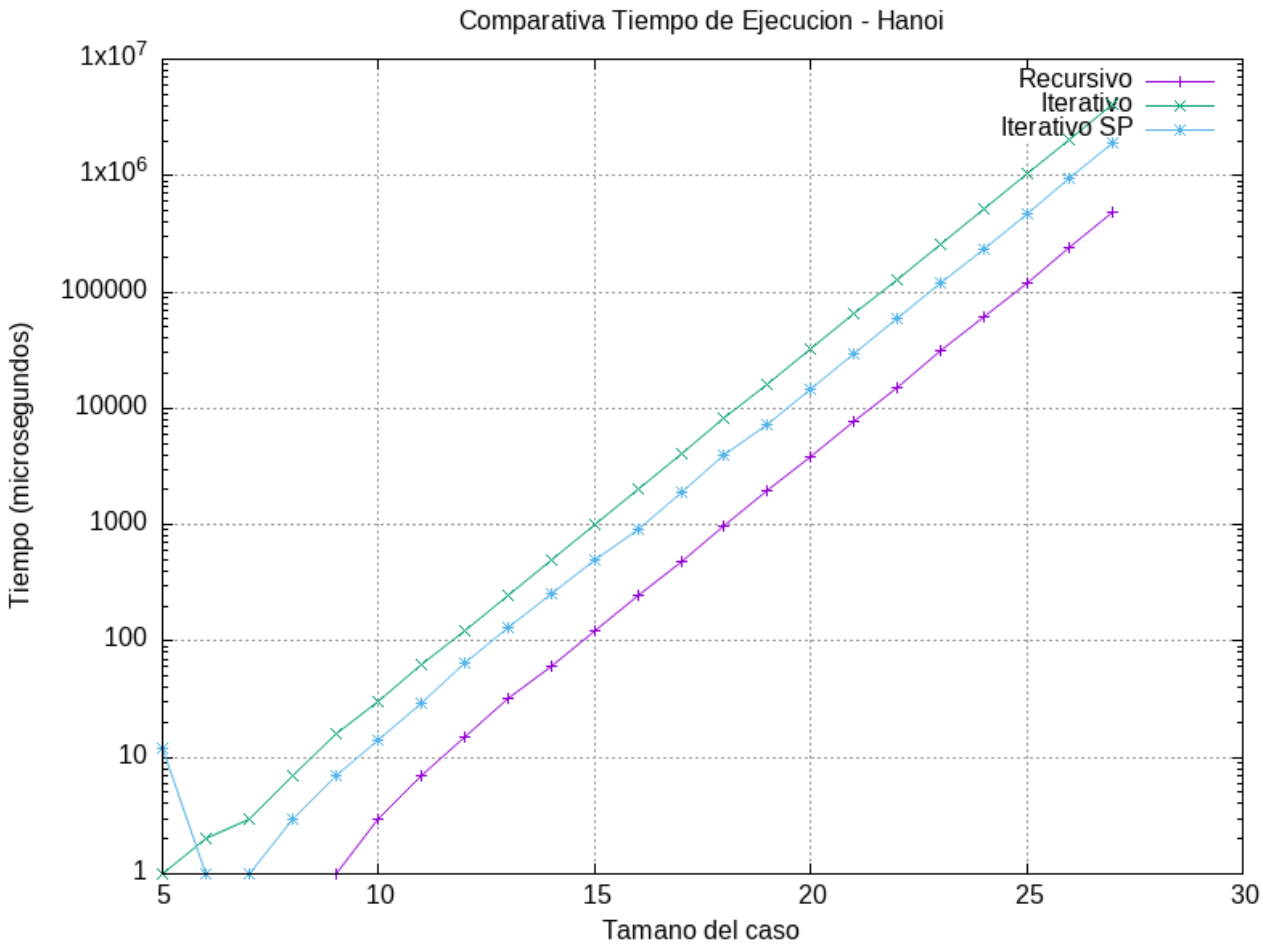
Ajuste por regresión de *Hanoi Iterativo Sin Pila*.

Comparación A continuación se presenta la comparación de los tres algoritmos de Hanoi:



Comparación de los algoritmos de Hanoi.

También la comparativa en escala logarítmica:



Comparación de los algoritmos de Hanoi en escala logarítmica.

3. Conclusión

En conclusión, tras realizar un estudio exhaustivo tanto desde un punto de vista teórico como empírico, hemos visto cómo influye el diseño algorítmico en la eficiencia de los programas informáticos.

Al analizar algoritmos de ordenación, observamos que QuickSort y MergeSort presentan una eficiencia significativamente superior a BubbleSort, especialmente cuando aumentamos el tamaño del conjunto de datos, lo que resulta en programas ejecutándose órdenes de magnitud más rápido.

En cuanto a los números de Catalan, la implementación recursiva mostró ser extremadamente ineficiente debido a su complejidad exponencial, haciendo impracticable su uso para valores elevados. Las versiones iterativas, especialmente la que usa coeficientes binomiales, destacaron por su rendimiento notablemente superior, demostrando así la importancia de técnicas avanzadas como la programación dinámica y la simplificación directa del cálculo.

Finalmente, al estudiar las Torres de Hanoi, constatamos que aunque todas las soluciones analizadas presentan una complejidad exponencial, las implementaciones iterativas mostraron ser ligeramente más rápidas en términos prácticos, ofreciendo pequeñas ventajas frente a la clásica solución recursiva.