UNIVERSIDAD DE GRANADA

Análisis de eficiencia de algoritmos

Algorítmica. Práctica 1

Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez
Sergio Quijano Rey
Lucía Salamanca López

segfault

Contenidos

Introducción

Análisis de eficiencia de algoritmos

- **Análisis de la eficiencia teórica:** predicción de clase de eficiencia.
- **Análisis de la eficiencia empírica:** ejecución y medición de tiempos.
- **Análisis de la eficiencia híbrida:** obtención de la constante oculta.

Cálculo de la eficiencia teórica

Consiste en analizar el peor tiempo de ejecución para decidir en qué clase de funciones en notación O grande se encuentra.

Cálculo de la eficiencia empírica

Hemos ejecutado los algoritmos y los hemos medido con `<chrono>`.

Hemos ejecutado cada etapa 100 veces y hemos hecho la media.

Cálculo de la eficiencia híbrida

Hemos procedido de dos maneras:

- Realizando la división $\frac{T(n)}{f(n)}$
- Ajustando con la curva de regresión

Análisis de algoritmos propuestos

Algoritmos analizados

En esta práctica hemos analizado los siguientes algoritmos:

1. Algoritmo de pivote
2. Algoritmo de búsqueda binaria (versión iterativa)
3. Algoritmo de búsqueda binaria (versión recursiva)
4. Algoritmo heapsort
5. Algoritmo bubble sort
6. Algoritmo mergesort
7. Algoritmo de las torres de Hanoi

Algoritmo de pivote

```
int pivotar(double *v, const int ini, const int fin) {
    double pivote = v[ini];
    double aux;
    int i = ini + 1, j = fin;

    while ( i <= j ) {
        while ( v[i] < pivote && i <= j ) i++;
        while ( v[j] == pivote && j >= i ) j--;
        if ( i < j ) {
            aux = v[i];
            v[i] = v[j];
            v[j] = aux;
        }
        if ( j < ini ) {
            v[ini] = v[j];
            v[j] = pivote;
        }
        return j;
    }
}
```

Algoritmo de pivote. Eficiencia teórica

Nos interesa estudiar el bucle externo y los dos internos.

Los `if` no afectan a los contadores. Su eficiencia es $O(1)$.

$$n = \texttt{j - i}$$

El bucle externo `while` se ejecuta n veces.

Los bucles internos incrementan el contador `i` y decrementan `j`.

La variación de contadores será de uno en uno $\Rightarrow$ los bucles internos son $O(1)$.

Al ejecutar el bucle n veces tenemos que:

$$T(n) \in O(n)$$

Binaria (versión iterativa)

```
int Busqueda(int *v, int n, int elem){
    int inicio = 0, fin = n-1, centro;

    centro = ( inicio + fin )/2;

    while ( ( inicio <= fin ) && ( v[centro] != elem ) ) {
        if ( elem < v[centro] )
            fin = centro - 1;
        else {
            inicio = centro + 1;
            centro = ( inicio + fin ) / 2;
        }
    }

    if( inicio > fin )
        return -1;

    return centro;
}
```

Búsqueda binaria (versión iterativa). Eficiencia teórica

Nos interesa el cuerpo del `while`. Sin pérdida de generalidad:

$$\texttt{inicio} = \frac{\texttt{inicio} + \texttt{fin}}{2}$$

Tomamos $n = \texttt{fin} - \texttt{inicio}$. En cada iteración, $n = \frac{n}{2}$.

Encontramos la iteración tal que $n_i \leq 1$

$$n_i = \frac{n}{2^i}$$

Despejando obtenemos que:

$$T(n) \in O(log_2 n)$$

Algoritmo de eliminación de repetidos

```
void EliminaRepetidos(double original[], int & nOiriginal){
    int i, j, k;

    for ( i = 0 ; i < nOiriginal ; i++ ) {
        j = i + 1;
        do {
            if ( original[j] == original[i] ){
                for( k = j+1 ; k < nOriginal ; k++ ){
                    original[k-1] = original[k];
                    nOriginal--;
                }
            } else {
                j++;
            }
        } while ( j < nOriginal );
    }
}
```

Algoritmo de eliminación de repetidos. Eficiencia teórica

Caso 1: Ningún elemento repetido

El bucle externo `for` se ejecuta n veces.

No entramos en el `if` $\Rightarrow$ `j` aumenta uno a uno.

Equivale a dos bucles `for` anidados:

$$T(n) \in O(n^2)$$

Algoritmo de eliminación de repetidos. Eficiencia teórica

Caso 2: Todos los elementos repetidos

Al entrar en el `if` siempre cada iteración del bucle `do while` decrementa en uno el tamaño.

El bucle `for` interno en cada iteración se realiza $n, n-1, \ldots, 1 \equiv \sum_{i=1}^{n-1} i$

$$T(n) \in O(n^2)$$

Algoritmo de eliminación de repetidos. Eficiencia teórica

Caso 3: Algunos elementos repetidos

Dividimos el vector en los dos casos anteriores, obteniendo así que:

$$T(n) \in O(n^2)$$

ya que ambos son $O(n^2)$

Algoritmo de eliminación de repetidos. Eficiencia teórica

Conclusión

Al haber estudiado los peores casos, podemos decir que:

$$T(n) \in O(n^2)$$

Búsqueda binaria (versión recursiva)

```
int BuscarBinario(int *v, const int ini, const int fin, const
    int x){

    int centro;

    if ( ini > fin )
        return -1;

    centro = ( ini + fin ) / 2;

    if ( v[centro] == x )
        return centro;

    if ( v[centro] > x )
        return BuscarBinario( v, ini, centro-1, x );

    return BuscarBinario( v, centro+1, fin, x );
}
```

Búsqueda binaria (versión recursiva). Eficiencia teórica

En la iteración m volvemos a llamar a la función con un tamaño $\frac{m}{2}$.

Ecuación de recurrencia:

$$T(m) = T(\frac{m}{2}) + 1$$

Sustituyendo m por 2^k y despejando, obtenemos:

$$T(2^k) - T(2^{k-1}) = 1$$

$$(x-1)^2 = 0$$

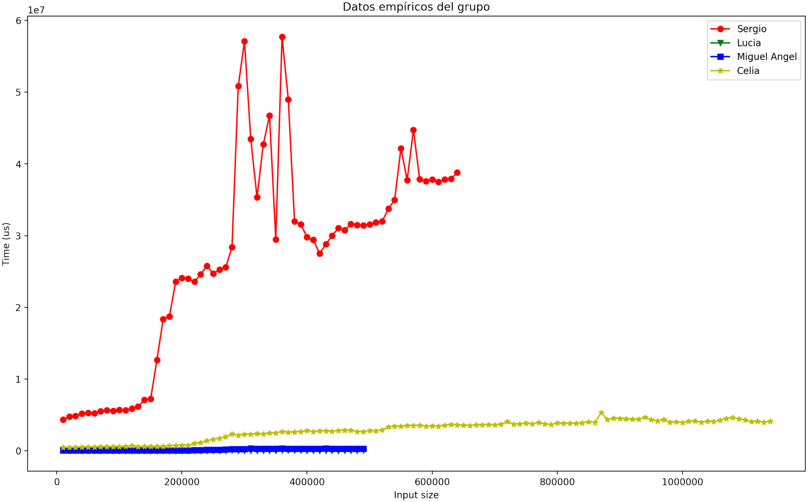
$$T(k) = c_1 1^k + c_2 k 1^k$$

$$T(n) = c_1 + c_2 log_2 n$$

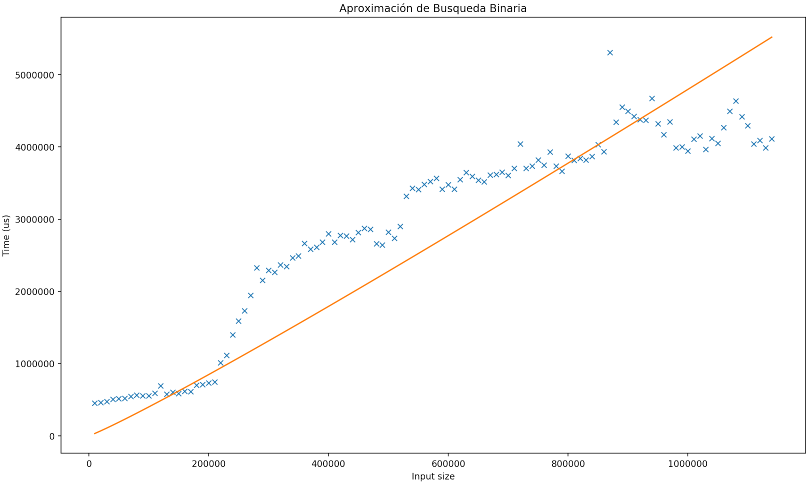
Así que:

$$T(n) \in O(log_2 n)$$

Búsqueda binaria (versión recursiva). Eficiencia empiríca



Búsqueda binaria (versión recursiva). Eficiencia híbrida



Búsqueda binaria (versión recursiva). Eficiencia híbrida

Constante oculta: $K = 0.8630419355569797$

Ajuste por regresión: $T(n) = 0.28166261 \cdot n \log n$

- **Error para recta (kn): 7.227564149512647 %**
- Error para cuadrática (kn^2): 56.14209694786215 %
- Error para cúbica (kn^3): 74.09603399513483 %
- Error para logarítmica ($k \log n$): 100.0 %
- **Error para n-logarítmica ($kn \log n$): 13.766671701121327 %**

Heap Sort

```
void heapsort(int T[], int num_elem) {
    for ( int i = num_elem/2; i >= 0; i-- ) {
        reajustar(T, num_elem, i);
    }
    for ( int i = num_elem-1; i >= 1; i-- ) {
        int aux = T[0];
        T[0] = T[i];
        T[i] = aux;
        reajustar(T, i, 0);
    }
}
```

Heap Sort

```
void reajustar(int T[], int num_elem, int k) {
    int j, v = T[k];
    bool esAPO = false;

    while ( ( k < num_elem/2 ) && !esAPO ) {
        j = 2*k + 1;
        if ( ( j < ( num_elem - 1 ) ) && ( T[j] < T[j+1] ) )
            j++;
        if ( v >= T[j] )
            esAPO = true;
        T[k] = T[j];
        k = j;
    }

    T[k] = v;
}
```

Heap Sort. Eficiencia teórica

Eficiencia de reajustar

Queremos el mayor número de iteraciones del bucle ($k < \frac{n}{2}$)

$$0 \leq k \leq n-1$$

No queremos que j incremente $\Rightarrow$ no entra en los `if`

$$k_i = 2k_{i-1} + c$$

Solución de la recurrencia:

$$k_i = 2^i - 1$$

Siendo el caso inicial $k_0 = 0$

Heap Sort. Eficiencia teórica

Eficiencia de reajustar

Queremos que $k = \frac{n}{2}$. Sustituimos

$$2^i - 1 = \frac{n}{2}$$

$$i = log_2 \frac{n-2}{2}$$

$$i = log_2 n - 2 - log_2 2$$

$$i = log_2 n - 2 - 1$$

Tenemos que

$$T(n) \in O(log_2(n))$$

Heap Sort. Eficiencia teórica

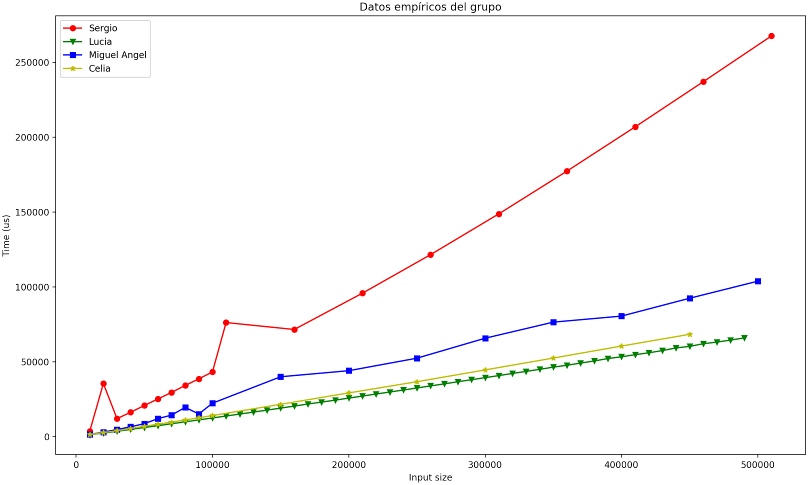
Eficiencia de heapsort

El primer `for` hace $\frac{n}{2}$ iteraciones y el segundo `for` hace $n - 1$ iteraciones

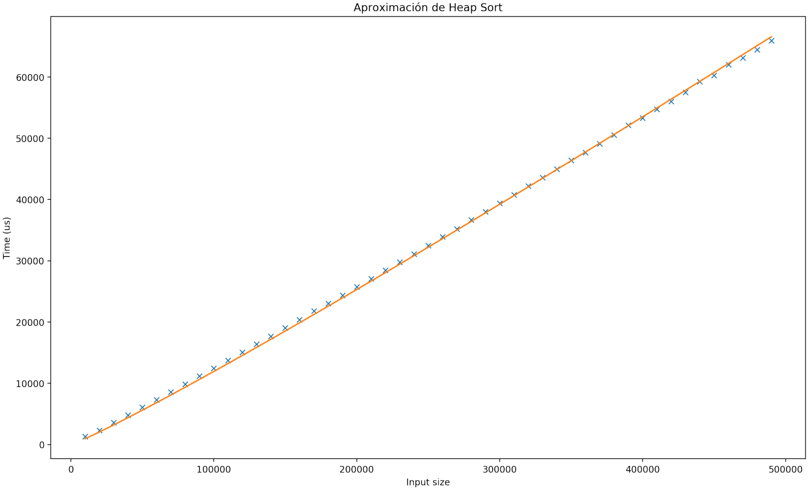
Como la eficiencia del interior del `for` es $O(log_2 n)$ tenemos que:

$$T(n) \in O(nlog_2 n)$$

Heap Sort. Eficiencia empírica



Heap Sort. Eficiencia híbrida



Heap Sort. Eficiencia híbrida

Constante oculta: $K = 0.9753044058661043$

Ajuste por regresión: $T(n) = 0.00718669 \cdot n \log n$

- **Error para recta (kn): 0.0403335419748782 %**
- Error para cuadrática (kn^2): 27.33530020306984 %
- Error para cúbica (kn^3): 74.09603399513483 %
- Error para logarítmica ($k \log n$): 100.0 %
- **Error para n-logarítmica ($kn \log n$): 0.0489468223025979 %**

Bubble Sort

```
void burbuja(int v[], int n) {
    int aux;

    for ( int i = inicial, i < final-1; i++ ) {
        for ( j = final-1; j > i; j-- ) {
            if ( v[j] < v[j-1] ) {
                aux = v[j];
                v[j] = v[j-1];
                v[j-1] = aux;
            }
        }
    }
}
```

Bubble Sort. Eficiencia teórica

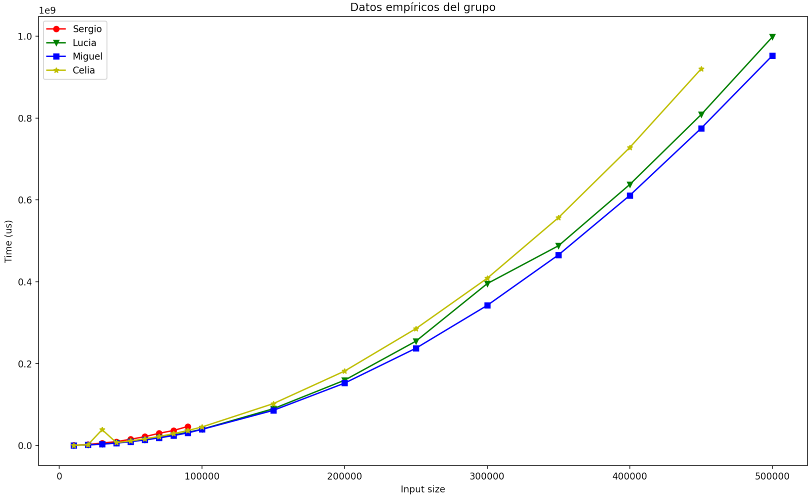
El `for` interno se ejecuta desde `fin-1` hasta `i` iteraciones y se ejecuta desde 0 hasta `fin-2`.

$$T(n) = \sum_{i=0}^{n-1} n - i - 1 = n^2 - \frac{(n-2)(n-1)}{2} - 5$$

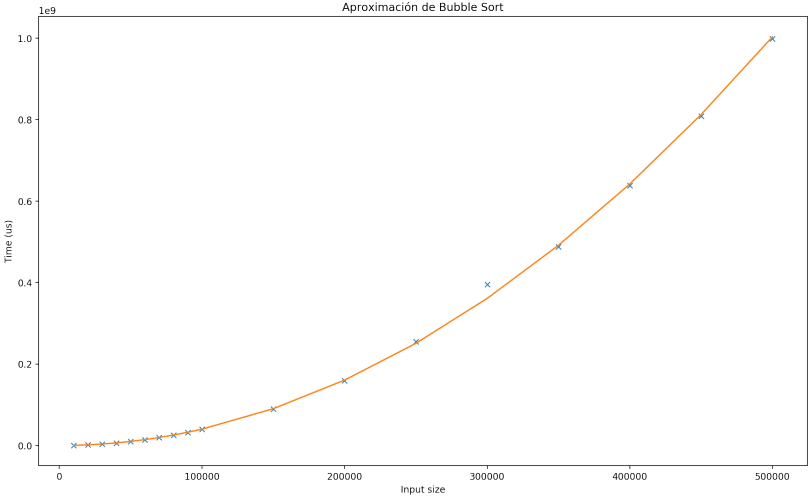
Por lo tanto

$$T(n) \in O(n^2)$$

Bubble Sort. Eficiencia empírica



Bubble Sort. Eficiencia híbrida



Bubble Sort. Eficiencia híbrida

Constante oculta: $K = 1.019659179330859$

Ajuste por regresión: $T(n) = 0.00401246 \cdot n^2$

- Error para recta (kn): 6.15853318137504 %
- **Error para cuadrática (kn^2): 0.08527050249560174 %**
- Error para cúbica (kn^3): 5.41027219494451 %
- Error para logarítmica ($k \log n$): 100.0 %
- Error para n-logarítmica ($kn \log n$): 10.096552530276018 %

Merge Sort

```
static void mergesort_lims(int T[], int inicial, int final) {
    if ( final-inicial < UMBRAL_MS )
        insercion_lims(T, inicial, final);
    else {
        int k = (final - inicial)/2;
        int * U = new int [k - inicial + 1];
        assert(U); int l, l2;
        for (l = 0, l2 = inicial; l < k; l++, l2++)
            U[l] = T[l2];
        U[l] = INT_MAX;
        int * V = new int [final - k + 1];
        assert(V);
        for (l = 0, l2 = k; l < final - k; l++, l2++)
            V[l] = T[l2];
        V[l] = INT_MAX;
        mergesort_lims(U, 0, k); mergesort_lims(V, 0, final-k);
        fusion(T, inicial, final, U, V);
        delete[] U; delete[] V;
    }
}
```

Merge Sort

```
static void fusion(int T[], int inicial, int final, int U[], int
     V[]) {
    int j = 0, k = 0;
    for ( int i = inicial; i < final; i++ ) {
        if (U[j] < V[k]) {
            T[i] = U[j];
            j++;
        } else {
            T[i] = V[k];
            k++;
        }
    }
}
```

Merge Sort. Eficiencia teórica

Eficiencia de fusión

Tiene un bucle `for` que se repite `final-inicial` por tanto:

$$T(n) \in O(n)$$

Merge Sort. Eficiencia teórica

Eficiencia de mergesort

Las dos llamadas a mergesort se hacen sobre vectores cuyo tamaño es la mitad que el original.

$$T(m) = 2T(\frac{m}{2}) + n$$

Sustituyendo m por 2^k y despejando:

$$T(2^k) - 2T(2^{k-1}) = 2^k$$

$$(x-2)^2 = 0$$

$$T(k) = c_1 * 2^k + c_2 * k * 2^k$$

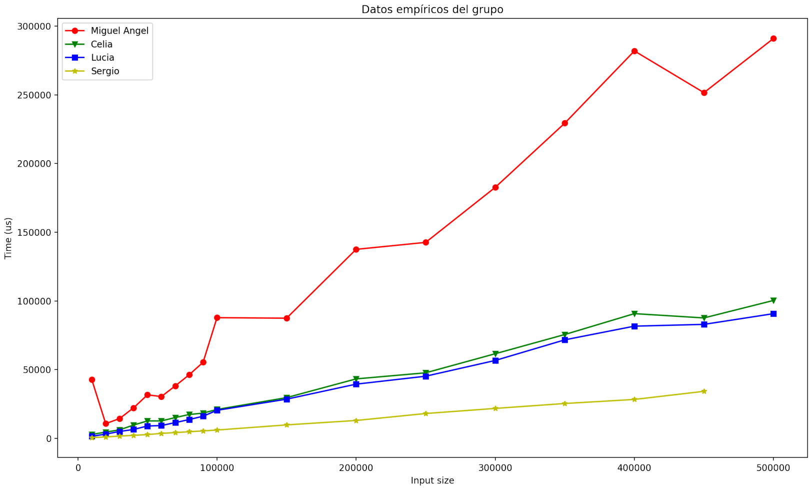
$$T(n) = c_1 * n + c_2 * log_2 n * n$$

Obtenemos que:

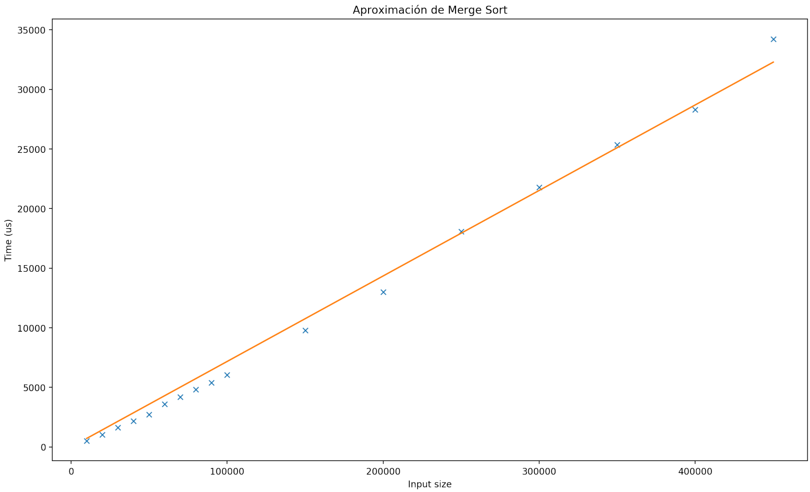
$$T(n) \in O(n log_2 n)$$

Observación: para tamaños pequeños se ejecutará inserción ($O(n^2)$)

Merge Sort. Eficiencia empírica



Merge Sort. Eficiencia híbrida



Merge Sort. Eficiencia híbrida

Constante oculta: $K = 1.1670955046281213$

Ajuste por regresión: $T(n) = 0.07175341 \cdot n \log n$

- Error para recta (kn): 0.3473246336299704 %
- Error para cuadrática (kn^2): 14.334551135315282 %
- Error para cúbica (kn^3): 37.228355082448616 %
- Error para logarítmica ($k \log n$): 100.0 %
- **Error para n-logarítmica ($kn \log n$): 0.246638483904476 %**

Hanoi

```
void hanoi(int M, int i, int j) {
    if ( M > 0 ) {
        hanoi(M-1, i, 6-i-j);
        hanoi(M-1, 6-i-j, j);
    }
}
```

Hanoi. Eficiencia teórica

Cada iteración el algoritmo se llama a sí mismo dos veces.

$$T(n) = 2T(n-1) + 1$$

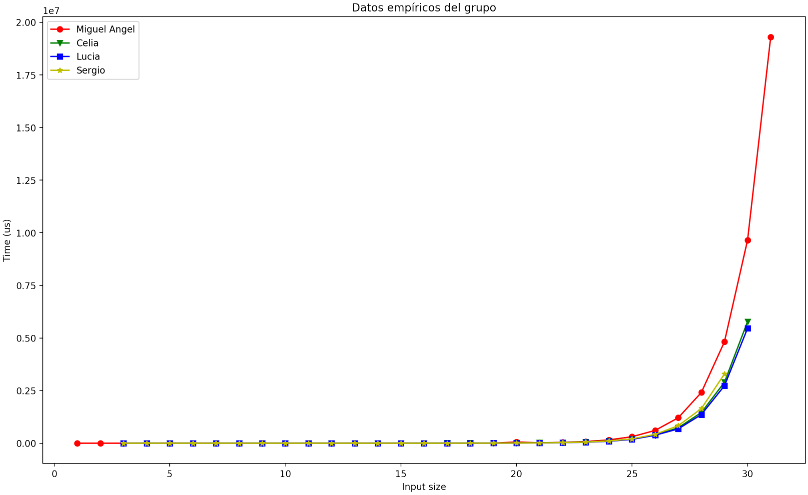
$$(x-2)(x-1) = 0$$

$$T(n) = c_1 * 2^n + c_2 * 1^n = c_1 * 2^n + c_2$$

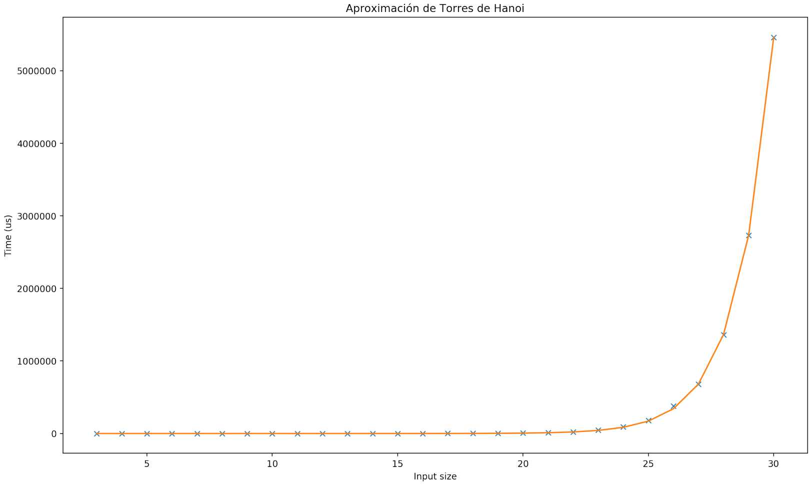
Por lo que:

$$T(n) \in O(2^n)$$

Hanoi. Eficiencia empírica



Hanoi. Eficiencia híbrida



Hanoi. Eficiencia híbrida

Constante oculta: $K = 0.7263436302553232$

Ajuste por regresión: $T(n) = 0.00508635 \cdot 2^n$

- Error para recta (kn): 76.01343903600339 %
- Error para cuadrática (kn^2): 71.13401593179258 %
- Error para cúbica (kn^3): 57.61256154034523 %
- Error para exponencial (ke^n): 3.1920070654956563 %
- **Error para potencial ($k2^n$): 0.004591299781178793 %**
- Error para logarítmica ($k \log n$): 100.0 %

Conclusión

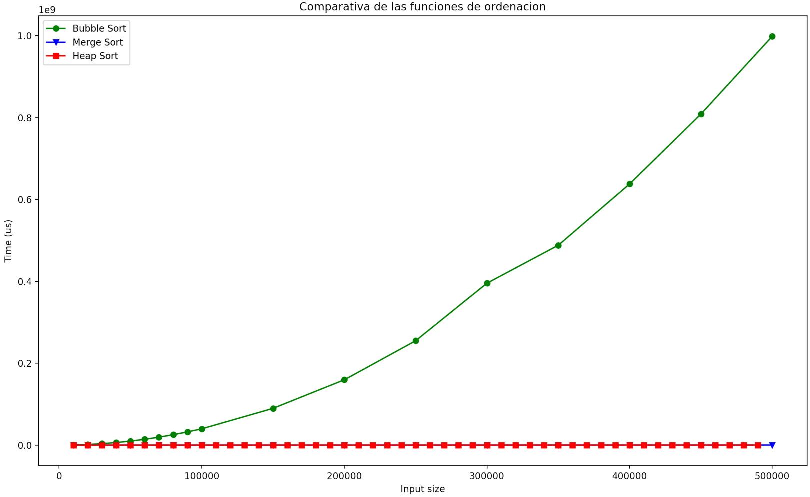
Conclusión

El análisis teórico ha sido correcto: correcta regresión.

Lo que más influye es la eficiencia del algoritmo.

Dependencia del computador: arquitecturas diferentes y prestaciones diferentes dan lugar a resultados diversos.

Comparativa entre algoritmos de búsqueda



Comparativa entre algoritmos de búsqueda