Actividad Académicamente Dirigida 2

- Grupo M2:
  - Ye Liu, Daniel Linfon (**representante**)
  - Bogacki Karwacki, Michal Marcin
  - Aponte Araujo, Alberto

- Enlace al video: **GrupoM2_AAD_2**

Índice de contenidos

Índice de contenidos

Ordenación por Selección

Resumen del algoritmo

- El algoritmo de **Ordenación por Selección** posiciona el elemento seleccionado en el lugar correcto dentro del vector. Para ello, se realiza una búsqueda en todo el vector para determinar cuál es el elemento menor, el cual una vez detectado se sitúa en la primera posición. A continuación, se busca el elemento más pequeño del array, excluyendo el que ya se ha posicionado, y se coloca en la segunda posición. Se procede de la misma forma sucesivamente, excluyendo de la búsqueda del elemento menor a aquellos que ya hemos posicionado, hasta completar la ordenación.

Algoritmo (pseudocódigo)

```
procedimiento selección(T[1..n])
    para i← 0 hasta n-2 hacer
        posminimo ← i
        para j ← i+1 hasta n-1 hacer
            si T[j] < T[posminimo] entonces
                posminimo ← j
            fsi
        fpara
        auxiliar ← T[posminimo]
        T[posminimo] ← T[i]
        T[i] ← auxiliar
    fpara
fprocedimiento
```

Ejemplo de ejecución / Traza.

Vector inicial: **n=5**

400, 200, 500, 300, 100

Primera pasada (i=1) hasta 4 (n-1):

400, 200, 500, 300, 100

Segunda pasada (i=2) hasta 4:

100, 200, 500, 300, 400

Tercera pasada (i=3) hasta 4:

100, 200, 500, 300, 400

Cuarta pasada (i=4) hasta 4:

100, 200, 300, 500, 400

ARRAY ORDENADO:

100, 200, 300, 400, 500

Leyenda

Amarillo: Posicion de i

Azul: Elemento menor

Verde: Elemento ordenado

Análisis de la eficiencia. Conclusiones (casos).

- **Caso mejor:** El número de comparaciones que realiza es de orden n^2, no hay intercambios.
- **Caso medio y peor:** El número de comparaciones es de orden n^2 y el de intercambios es de orden n.

$$T(n) = O(n^2) \text{ Para todos los casos}$$

$$T(n) = \frac{n^2 - n}{2}$$

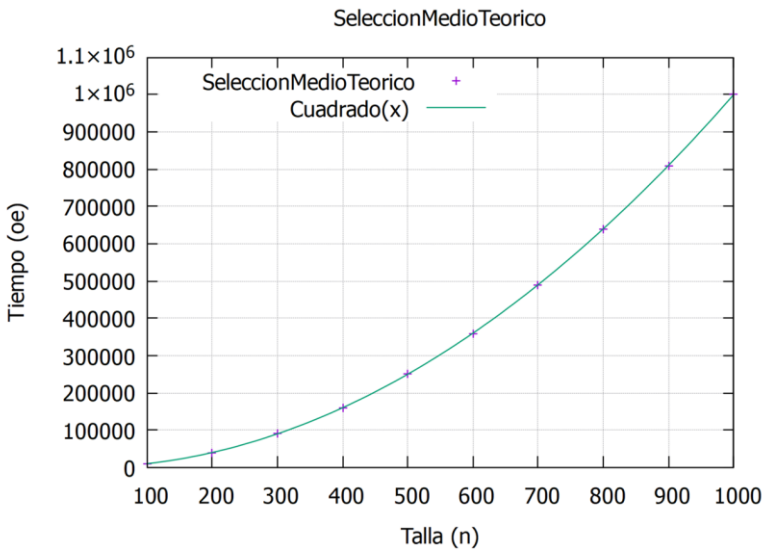
Algoritmo (código en C++).

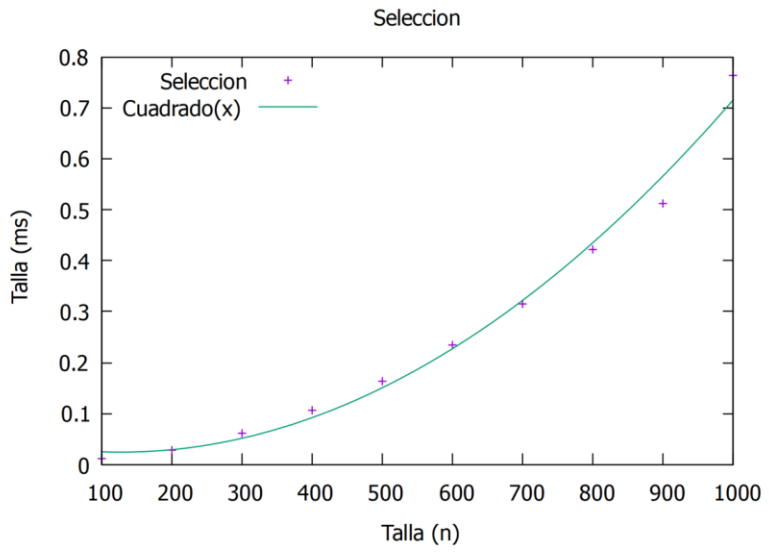
```
void ordenaSeleccion(int v[], int size)
{
    int posminimo, aux;
    for (int i = 0;i < size - 1;i++)
    {
        posminimo = i;
        for (int j = i + 1;j < size ;j++)
        {
            if (v[j] < v[posminimo])
                posminimo = j;
        }
        aux = v[posminimo];
        v[posminimo] = v[i];
        v[i] = aux;
    }
}
```

Gráficas de coste teóricas y empíricas. Conclusiones.

Teórico:

Empírica:





Tablas obtenidas gracias a un programa de la casa que implementa los proyectos de las prácticas 1 y 2

Ordenación por Sacudidas(Shakersort)

Resumen del algoritmo

- El funcionamiento de este algoritmo consiste en recorrer el vector intercambiando los elementos adyacentes que aún estén en desorden, dependiendo de si esta se realiza de manera ascendente o descendente.

- En cada tirada usará el mayor valor y lo dispondrá en la posición derecha máxima del vector desordenado (en ascenso), una vez situado en la posición correspondiente, disminuirá una posición el valor máximo del vector, luego descenderá y hará que se posicione el menor valor en el extremo izquierdo que está desordenado, e incrementará el valor mínimo del vector.

- Por cada tirada, el vector se reducirá en dos posiciones, por ello se considera una mejora del algoritmo Burbuja ya que se ejecuta la mitad de veces que este.

- A medida que desplaza los elementos se reduce el tamaño del vector desordenado, hasta que queda totalmente ordenado.

Algoritmo pseudocódigo

```
 Algoritmo ShakerSort (datos, n; var N)
          Entrada: Un array datos que almacena n enteros. Un N entero.
Comienzo
Enteros posi_vect_izq ← 0, posi_vect_dcha ← N-1, i, j, temporal;
Mientras (posi_vect_izq <= posi_vect_dcha) entonces
        Para i ← posi_vect_izq hasta i < posi_vect_dcha hacer
                  Si datos [i] > datos [i+1] entonces
                  temporal ← datos [i]
                  datos[i] ← datos [i+1]
                  datos [i+1] ← temporal
                F_Si
         F_Para
         Posi_vect_dcha ← posi_vect_dcha –1;
         Para j ← posi_vect_dcha hasta j > posi_vect_izq hacer
                  Si datos [j] < datos [j-1] entonces
                  temporal ← datos [j];
                  datos [j] ← datos [j-1];
                  datos [j-1] ← temporal;
               F_si
       F_Para
        Posi:vector_izq ← posi_vect_izq +1;
F_Mientras
```

Ejemplo de ejecución/Traza

- Por cada iteración del bucle **Mientras** sucede:

  - El método compara los elementos adyacentes, una vez que ordena el elemento disminuye el tamaño del vector.

    Vector base: **350, 380, 890, 900, 700, 500.**

  - En la primera pasada, el mayor hacia la derecha y el menor hacia la izquierda, entonces el vector resultante tiene el primer y el último elemento ordenado, lo que resulta en que el tamaño es de cuatro elementos y se vuelve a ejecutar el bucle Mientras:

    350, 380, 500, 890, 700, 900.

  - Tras otra pasada el vector quedaría ordenado, aunque se ejecutaría una última vez, ya que las variables que controlan el bucle son dos contadores, pero no se ejecutarán las sentencias dentro de los If porque la condición ya no se cumple, quedando el vector como:

    350,380, 500, 700, 890, 900.

Análisis de la eficiencia. Conclusiones.

- El **mejor caso** para este algoritmo ShakerSort, se dará cuando el vector esté previamente ordenado de forma ascendente.

  T(n)= 4n^2-6n-7

- El **peor caso** se dará cuando esté ordenado de manera descendente, al ejecutarse ambas operaciones de los Si.

  T(n)= (34/4)n^2 - (39/2)n - 34

- Y en el **caso medio**, el vector estará ordenado aleatoriamente., en nuestro caso supondremos que el cuerpo del if se ejecutará la mitad de las veces.

  T(n)= (50/8)n^2 - (51/4)n - 41/2

$$T(n) = O(n^2) \text{ Para todos los casos}$$

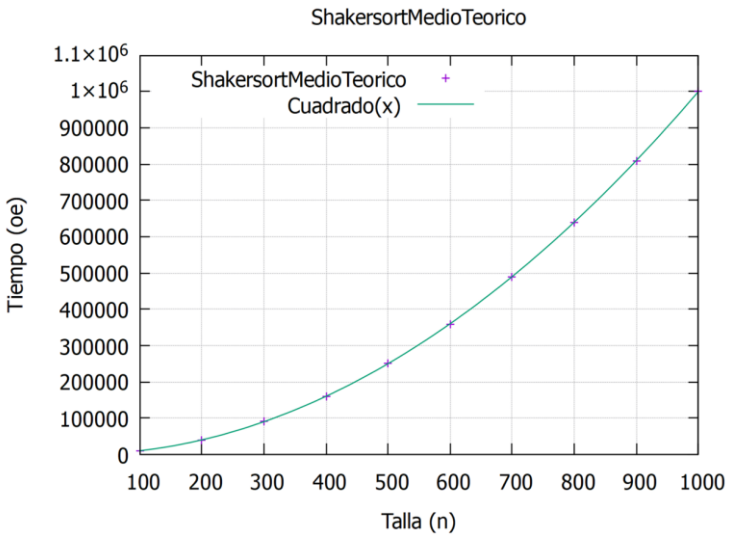
Algoritmo(código en C++).

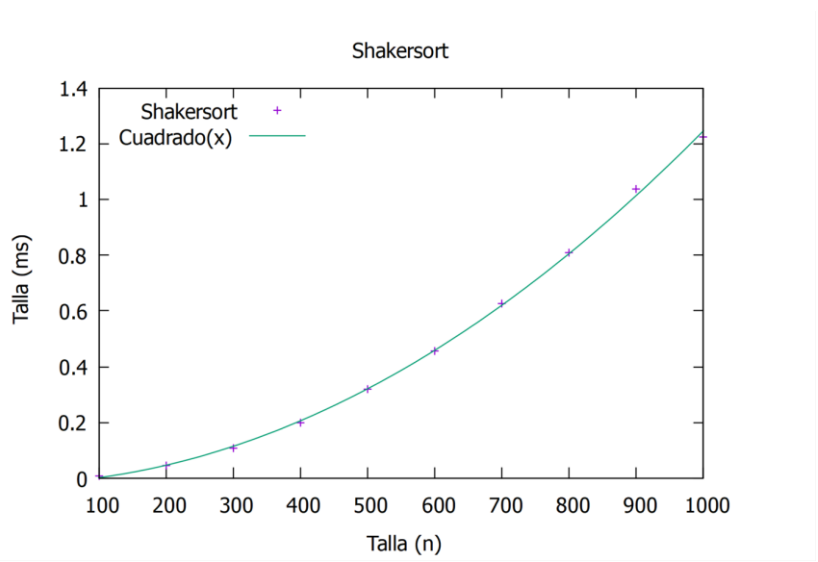
```
void ordenaShakerSort(int v[], int size)
{
	int N = size;
	int posi_vector_izq = 0;
	int posi_vector_dcha = N-1;
	while (posi_vector_izq <= posi_vector_dcha){
		for (int i = posi_vector_izq; i < posi_vector_dcha; i++){
				if (v[i] > v[i+1]){
						int temporal = v[i];
					v[i] = v[i+1];
					v[i+1] = temporal ;
				}
		}
				posi_vector_dcha--;
				for (int j = posi_vector_dcha; j > posi_vector_izq; j--)  {
				if (v [j] < v[j-1])  {
						int temporal = v[j]
						v[j] = v[j-1];
						v[j-1] = temporal;
				}
				}
		posi_vector_izq++;
		 }
}
```

Gráficas de coste teóricas y empíricas. Conclusiones.

Teórico:

Empírica:





Tablas obtenidas gracias a un programa de la casa que implementa los proyectos de las prácticas 1 y 2

Ordenación por Mezcla (Mergesort)

Resumen del algoritmo

- El algoritmo de **Ordenación por Mezcla** se basa en el tópico “dividir para vencer”. En este método se *dividen* los elementos en 2 arrays de misma longitud, aproximadamente. Acto seguido se *ordena* independientemente cada secuencia. Y por último, se *mezclan* los vectores de la siguiente manera: selecciona el menor de los elementos de las dos secuencias, lo añade a la secuencia ordenada final y se elimina de su antiguo array. Cuando uno de los array quede vacío, se concatena en la secuencia final el array en el que queden elementos.

Algoritmo (pseudocódigo)

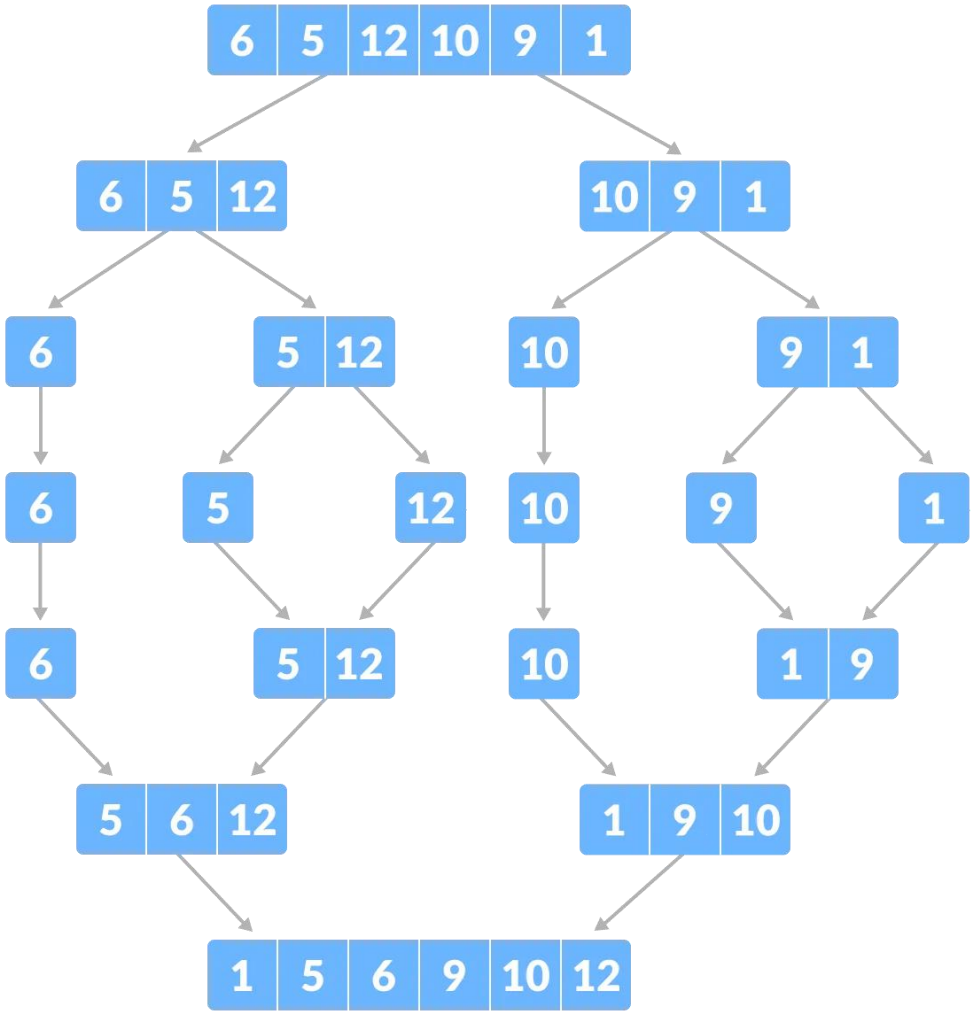
```
Procedimiento MergeSort(A, p, r) /*Ordena un vector A desde p hasta r*/
si p<r entonces
/* dividir en dos trozos de tamaño igual (o lo más parecido posible), es decir [n/2] y [n/2]*/

    q ← (p+r)/2;                                  /* Divide */
    /* Resolver recursivamente los subproblemas */
    MergeSort (A,p,q);                            /* Resuelve */
    MergeSort (A,q+1,r);                          /* Resuelve */
    /* Combinar: mezcla dos listas ordenadas en (n) */
    Merge (A,p,q,r);                              /* Combina*/
```

- **Llamada inicial: MergeSort(A,1,n), siendo n el número de elementos del vector A**

Algoritmo (pseudocódigo)

```
Procedimiento Merge(A, p, q, r)
     i←p; j←q+1; k←0;
        mientras (i <= q && j <= r) entonces
                si (A[i] <= A[j]) entonces
                        B[k++] = A[i++];
                f_si
                        B[k++] = A[j++];
                k++;
        mientras(i <= q) entonces
                B[k++] = A[i++];
        mientras (j <= r) entonces
                B[k++] = A[j++];
        para k ← p hasta r hacer
                A[k] = B[k-p];
```

Ejemplo de ejecución / Traza.



Análisis de la eficiencia. Conclusiones (casos).

- **Caso mejor:** Se dará cuando el vector inicial esté ordenado de menor a mayor valor.
- **Caso medio y peor:**
  - ***Dividir: D(n) = O(1),***
  - ***Vencer:*** Como dividimos el problema a la mitad y lo resolvemos recursivamente, entonces estamos resolviendo dos problemas de tamaño ***n/2*** lo que nos da un tiempo ***T(n) = 2T(n/2)***
  - ***Combinar:*** Merge (fusión) de un vector de n elementos que es de orden lineal, por lo que tenemos que ***C(n)=O(n)***

Para sustituir, sumamos ***D(n) + C(n) = O(n) + O(1) = O(n)*** , por lo que sustituyendo en la recurrencia tenemos que:

T(n) = O(nlog(n)) Para todos los casos

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

Algoritmo (código en C++).

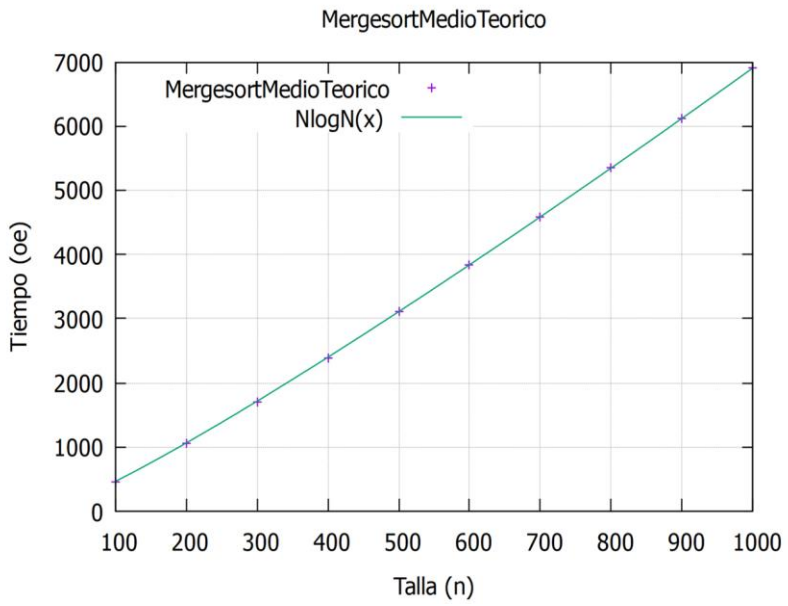
```
void ordenaMergesort(int v[], int size)
{
        mergesort(v, 0, size - 1);
}
void mergesort(int v[], int e, int d)
{
        if (e < d)
        {
                int m = (e + d) / 2;
                mergesort(v, e, m);
                mergesort(v, m + 1, d);
                merge(v, e, m, d);
        }
}
```

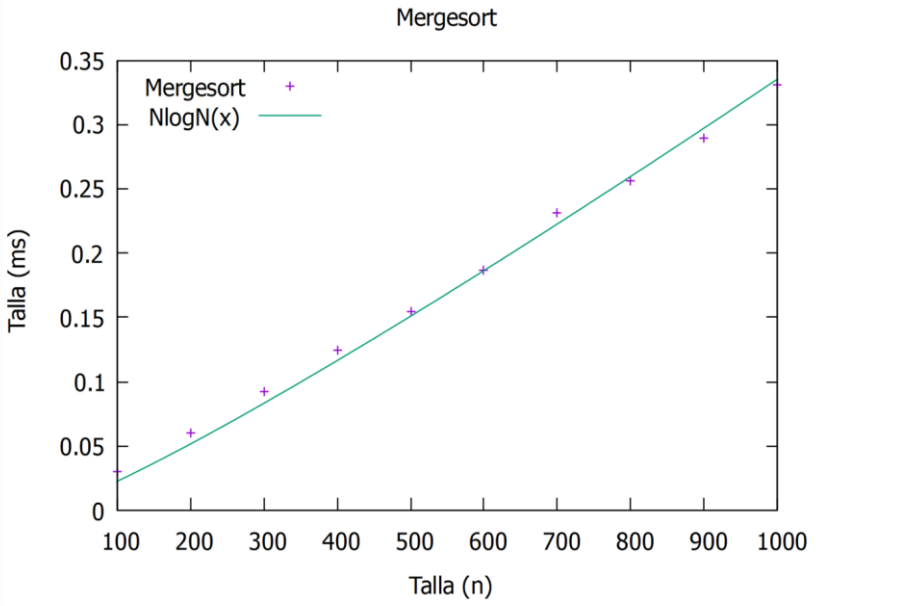
Algoritmo (código en C++).

```
void merge(int v[], int e, int m, int d)
{
        int *B = new int[d - e + 1];
        int i = e; int j = m + 1; int k = 0;
        while (i <= m && j <= d) {
                if (v[i] < v[j])
                        B[k++] = v[i++];
                else
                        B[k++] = v[j++];
        }
        while(i <= m)
                B[k++] = v[i++];
        while (j <= d)
                B[k++] = v[j++];
        for (k = 0; k <= d - e; ++k)
                v[e + k] = B[k];
        delete[] B;
}
```

Gráficas de coste teóricas y empíricas. Conclusiones.

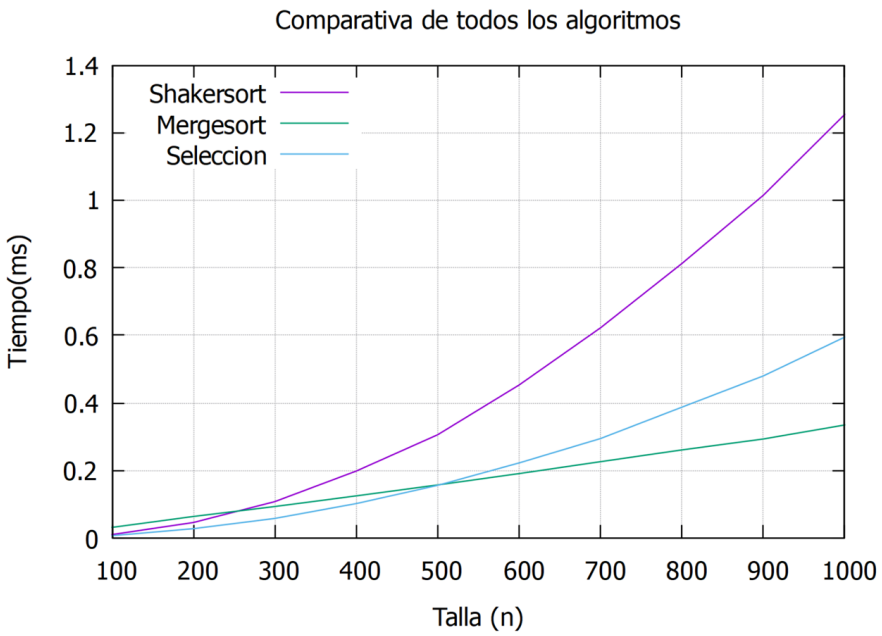
Teórico:
Empírica:





Tablas obtenidas gracias a un programa de la casa que implementa los proyectos de las prácticas 1 y 2

Conclusiones de la actividad: comparación de los algoritmos estudiados.



La gráfica muestra cómo los diferentes algoritmos hacen uso del **mismo tiempo** para **tallas bajas**. Sin embargo, a medida que avanzan las funciones, vemos cómo para **tallas elevadas** el algoritmo **Mergesort** es el más **rápido**, frente al **Shakersort** que es el más **lento**. Sobre las **trazas** decir que tanto el método **Selección** como el **Shakersort** crecen de manera **cuadrática**, a diferencia del **Mergesort** que, pese a ser una función **logarítmica**, crece de una forma más lineal. Con esto concluimos que el algoritmo más **óptimo** entre los 3 es el ***Mergesort***.

Bibliografía. Referencias web.

- Apuntes Tema 5 y 6 FAA

- Enlaces web
  - Ordenamiento de burbuja bidireccional
  - Rendimiento de algoritmos y notación Big-O
  - Mergesort
  - Selección
- Programas
  - Proyecto propio “ProgramaGráficas”: