Análisis de Algoritmos 2022/2023

Práctica 2

Grosso Christian, Ștefan Gabriel, Grupo 8.

Código	Gráficas	Memoria	Total

1. Introducción.

En la práctica anterior se desarrollaron rutinas para la creación de arreglos con permutaciones aleatorias, que luego se ordenaban utilizando algoritmos de ordenamiento como Bubble Sort y su versión mejorada, Bubble Sort Flag. Posteriormente, se recopilaron, a través de la biblioteca time.c, los tiempos promedio de ordenamiento de las permutaciones y el número de operaciones básicas ejecutadas (mínimas, máximas y promedio). Con estos datos se generaron gráficos para comparar la ejecución de los algoritmos con las predicciones teóricas y verificar su correspondencia. En esta práctica, se realizará esencialmente lo mismo, pero con dos nuevos algoritmos de ordenamiento: Merge Sort y Heap Sort.

2. Objetivos

2.1 Apartado 1

Crear una rutina que implemente el algoritmo de ordenación *MergeSort* sobre una permutación. La rutina deberá recibir como entradas un **array** que contiene la permutación y los índices del primer (**ip**) y último (**iu**) elemento de la permutación. Devolverá el número de veces que se ha ejecutado la operación básica (OB).

2.2 Apartado 2

Modificando el programa exercise5.c de la práctica anterior para utilizar el algoritmo *MergeSort*, obtener la tabla correspondiente a la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Representad dichos valores y compararlos con el resultado teórico del algoritmo

2.3 Apartado 3

Crear una rutina que implemente el algoritmo de ordenación *heapsort* sobre una permutación. La rutina deberá recibir como entradas un **array** que contiene la permutación y los índices del primer (**ip**) y último (**iu**) elemento de la permutación. Devolverá el número de veces que se ha ejecutado la operación básica (OB). También crear las rutinas CrearHeap, OrdenarHeap y Heapify, necesarias para realizar una ordenación eficiente

2.4 Apartado 4

Modificando el programa exercise5.c de la práctica anterior para utilizar el algoritmo *heapsort*, obtener la tabla correspondiente a la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Representad dichos valores y compararlos con el resultado teórico del algoritmo

3. Herramientas y metodología

El ambiente de desarrollo utilizado es el Subsistema de Windows para Linux (**WSL**). Otras herramientas utilizadas para el desarrollo del código fueron **Visual Studio Code** y la **extensión GCC** para Visual Studio Code. Para la parte de los gráficos, utilicé **Gnuplot** con las guías disponibles en Moodle. Las pruebas del código fueron principalmente las proporcionadas por el profesor en el fichero "**MAKEFILE**", en algunos casos modificando los parámetros para verificar los casos límite y el retorno de los errores.

3.1 Apartado 1

El algoritmo MergeSort se implementó utilizando el enfoque de Divide y Vencerás, dividiendo recursivamente el arreglo en subarreglos más pequeños, ordenándolos y fusionándolos nuevamente. La función de fusión combina dos subarreglos ordenados en un único arreglo ordenado mientras realiza un seguimiento de las operaciones básicas y gestiona dinámicamente la memoria. El manejo de errores asegura la robustez durante fallos en la asignación de memoria. Esta implementación ordena arreglos de manera eficiente con una complejidad temporal de O(n log n) y muestra el total de operaciones básicas realizadas.

Además, el programa principal se modificó para probar el algoritmo MergeSort. El tamaño del arreglo se especifica mediante argumentos de línea de comandos, y se genera una permutación aleatoria utilizando generate_perm. El resultado ordenado se muestra para verificar la corrección, y un manejo adecuado de errores asegura la robustez durante la ejecución.

3.2 Apartado 2

El programa evalúa el rendimiento de MergeSort midiendo los tiempos de ejecución promedio, mínimo y máximo, así como las operaciones básicas, para diferentes tamaños de entrada. Los argumentos de línea de comandos especifican el rango de tamaño de entrada (-num_min, -num_max), el incremento (-incr), el número de permutaciones (-numP) y el archivo de salida (-outputFile). La función generate_sorting_times calcula y escribe los resultados en el archivo, permitiendo la comparación con la complejidad teórica O(n log n) Un manejo robusto de errores asegura una ejecución adecuada y la validación de entradas.

3.3 Apartado 3

HeapSort se implementó construyendo un montículo máximo (max-heap) y extrayendo el mayor elemento para ordenar el arreglo. La función Heapify asegura la propiedad del montículo ajustando intercambios recursivamente. CrearHeap construye el montículo máximo aplicando Heapify a nodos no hoja, y OrdenarHeap ordena extrayendo la raíz y re-aplicando Heapify. Se registra el número de operaciones básicas para evaluar el rendimiento.

El programa principal fue adaptado para probar HeapSort, especificando el tamaño del arreglo con argumentos de línea de comandos y generando permutaciones aleatorias con generate_perm. Se verifica la corrección del orden y se manejan errores para garantizar robustez.

3.4 Apartado 4

El programa mide el rendimiento de HeapSort calculando tiempos promedio, mínimo y máximo, y operaciones básicas para distintos tamaños de entrada. Los argumentos de línea de comandos configuran el rango (-num_min, -num_max), incremento (-incr), permutaciones (-numP) y archivo de salida (-outputFile). generate_sorting_times guarda los resultados, comparados con la complejidad teórica O(nlog⁡n)O(n\log n)O(nlogn). Se garantiza robustez con manejo de errores.

4. Código fuente

4.1 Apartado 1

```
// Merge function: Combines two sorted subarrays into one sorted array
int merge(int* tabla, int ip, int iu, int imedio) {
    int basic_operations = 0;
    // Sizes of the two subarrays
    int n1 = imedio - ip + 1;
    int n2 = iu - imedio;
    // Temporary arrays for the two subarrays
    int* left = (int*)malloc(n1 * sizeof(int));
    int* right = (int*)malloc(n2 * sizeof(int));
    if (!left || !right) {
        // Memory allocation failed
        return ERR;
    }
    // Copy data to temporary arrays
    for (int i = 0; i < n1; i++) {
        left[i] = tabla[ip + i];
        basic_operations++;
    }
    for (int i = 0; i < n2; i++) {
        right[i] = tabla[imedio + 1 + i];
        basic_operations++;
    }
    // Merge the two temporary arrays back into `tabla`
    int i = 0, j = 0, k = ip;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            tabla[k++] = left[i++];
        } else {
            tabla[k++] = right[j++];
        }
        basic_operations++;
    }
    // Copy any remaining elements of `left`
    while (i < n1) {
        tabla[k++] = left[i++];
        basic_operations++;
    }
    // Copy any remaining elements of `right`
    while (j < n2) {
        tabla[k++] = right[j++];
        basic_operations++;
    }
    // Free temporary arrays
    free(left);
    free(right);
    return basic_operations;
}
```

```
// MergeSort function: Recursively sorts an array
int mergesort(int* tabla, int ip, int iu) {
    int basic_operations = 0;
    if (ip < iu) {
        int imedio = ip + (iu - ip) / 2;

        // Sort the first half
        int left_operations = mergesort(tabla, ip, imedio);
        if (left_operations == ERR) {
            return ERR;
        }
        basic_operations += left_operations;

        // Sort the second half
        int right_operations = mergesort(tabla, imedio + 1, iu);
        if (right_operations == ERR) {
            return ERR;
        }
        basic_operations += right_operations;

        // Merge the two halves
        int merge_operations = merge(tabla, ip, iu, imedio);
        if (merge_operations == ERR) {
            return ERR;
        }
        basic_operations += merge_operations;
    }

    return basic_operations;
}
```

4.3 Apartado 3

```c
int Heapify(int* tabla, int n, int i) {
    int basic_operations = 0;

    int largest = i;           // Initialize largest as root
    int left = 2 * i + 1;      // Left child
    int right = 2 * i + 2;     // Right child

    // Check if left child exists and is greater than root
    if (left < n && tabla[left] > tabla[largest]) {
        largest = left;
    }
    basic_operations++;

    // Check if right child exists and is greater than largest so far
    if (right < n && tabla[right] > tabla[largest]) {
        largest = right;
    }
    basic_operations++;

    // If largest is not root, swap and continue heapifying
    if (largest != i) {
        int temp = tabla[i];
        tabla[i] = tabla[largest];
        tabla[largest] = temp;
        basic_operations++;

        // Recursively heapify the affected subtree
        int sub_operations = Heapify(tabla, n, largest);
        if (sub_operations == ERR) {
            return ERR;
        }
        basic_operations += sub_operations;
    }

    return basic_operations;
}
```

```c
int CrearHeap(int* tabla, int n) {
    int basic_operations = 0;

    // Start from the last non-leaf node and heapify each node
    for (int i = n / 2 - 1; i >= 0; i--) {
        int operations = Heapify(tabla, n, i);
        if (operations == ERR) {
            return ERR;
        }
        basic_operations += operations;
    }

    return basic_operations;
}

//Description: Performs heap sort
int OrdenarHeap(int* tabla, int n) {
    int basic_operations = 0;

    // Create a heap from the array
    int heap_operations = CrearHeap(tabla, n);
    if (heap_operations == ERR) {
        return ERR;
    }
    basic_operations += heap_operations;

    // Extract elements one by one from the heap
    for (int i = n - 1; i > 0; i--) {
        // Move the root (largest element) to the end
        int temp = tabla[0];
        tabla[0] = tabla[i];
        tabla[i] = temp;
        basic_operations++;

        // Call Heapify on the reduced heap
        int operations = Heapify(tabla, i, 0);
        if (operations == ERR) {
            return ERR;
        }
        basic_operations += operations;
    }

    return basic_operations;
}
```

```c
// Wrapper HeapSort
int heapsort(int* tabla, int ip, int iu) {
    if (ip >= iu) {
        return 0;  // No sorting needed for a single element
    }

    // Calculate the size of the array
    int n = iu - ip + 1;

    // Allocate a temporary array for the range ip to iu
    int* temp = (int*)malloc(n * sizeof(int));
    if (!temp) {
        return ERR;
    }

    // Copy the relevant range to the temporary array
    for (int i = 0; i < n; i++) {
        temp[i] = tabla[ip + i];
    }

    // Perform heap sort on the temporary array
    int basic_operations = OrdenarHeap(temp, n);
    if (basic_operations == ERR) {
        free(temp);
        return ERR;
    }

    // Copy the sorted array back to the original array
    for (int i = 0; i < n; i++) {
        tabla[ip + i] = temp[i];
    }

    free(temp);
    return basic_operations;
}
```

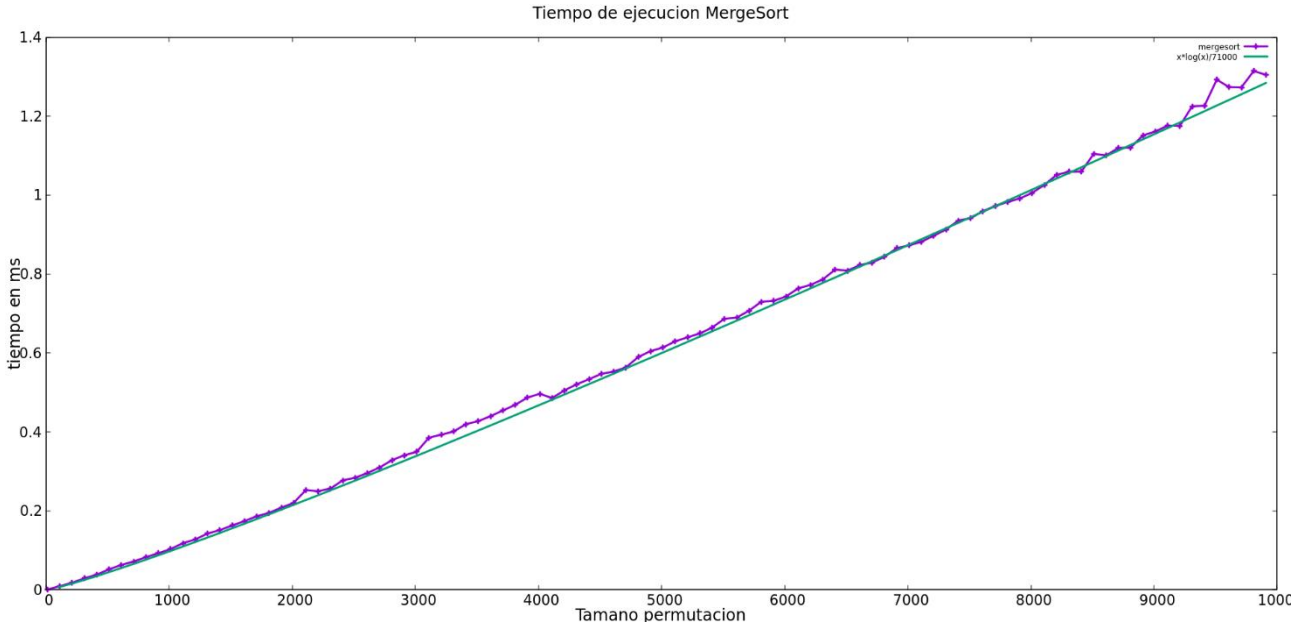
5. Resultados, Gráficas

5.1 Apartado 1

La función ordena correctamente una permutación.

5.2 Apartado 2

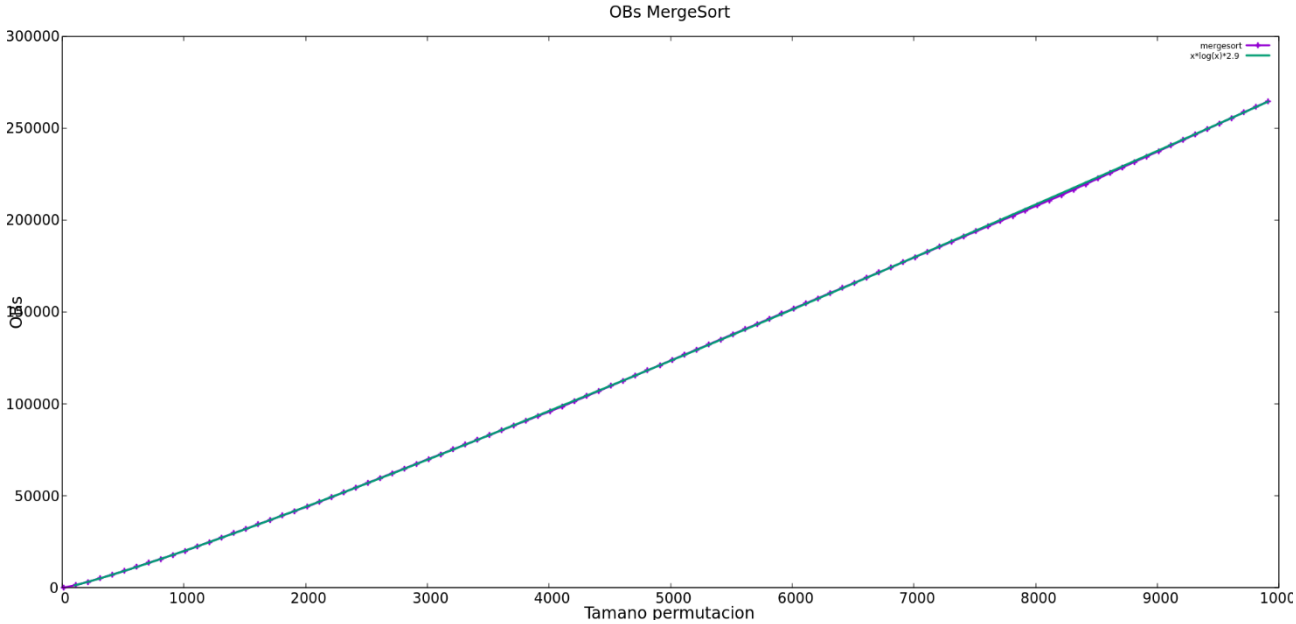
Tiempo



Del gráfico observamos el comportamiento del tiempo de ejecución de MergeSort en función del tamaño de la permutación. Los resultados parecen compararse con una función teórica (xlog(x)) normalizada por un factor $\frac{1}{71000}$ que representa la complejidad esperada de MergeSort en O(nlog n).

En teoría, MergeSort tiene una complejidad temporal de O(n log n). Esto significa que el tiempo de ejecución crece de manera sublineal en comparación con n^2, pero más rápidamente que un crecimiento lineal O(n).

Operaciones básicas



El gráfico muestra que el comportamiento empírico de las Operaciones Básicas de MergeSort es coherente con las previsiones teóricas O(n log n)).

La curva xlog(x) representa con precisión el número de operaciones básicas, confirmando la eficiencia y la corrección de la implementación práctica respecto a la complejidad teórica.

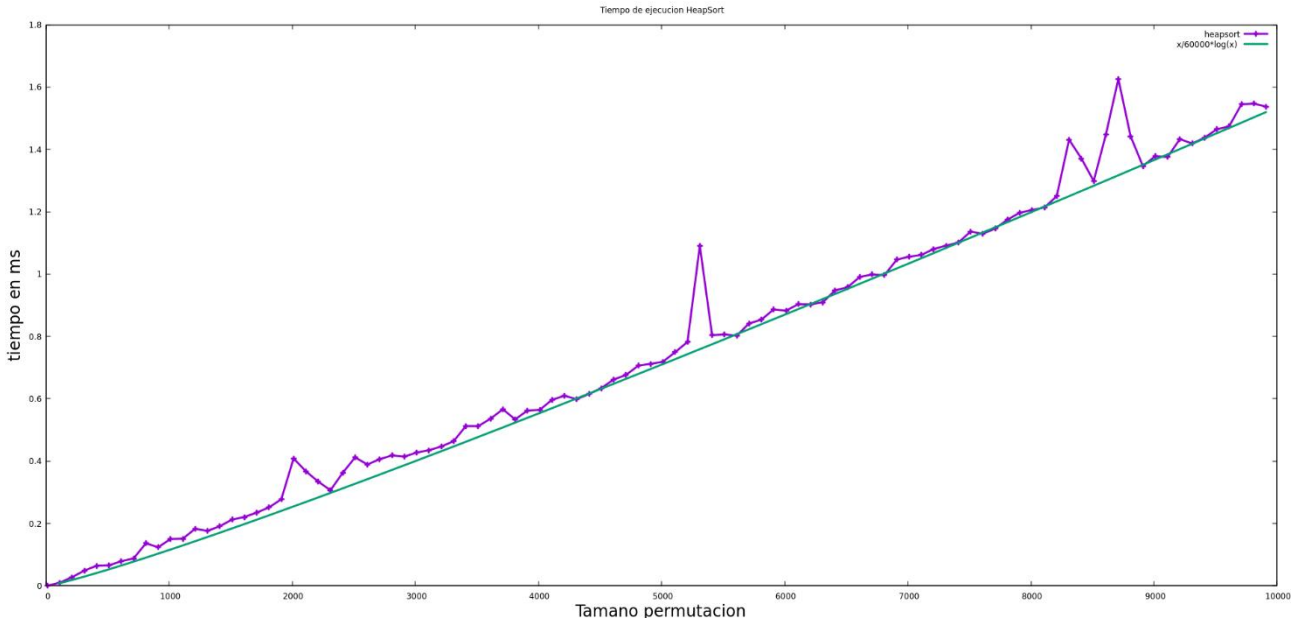
El número total de operaciones básicas teóricas puede aproximarse mediante la función: $OB(n)=c \cdot n \cdot \log(n)$, donde c representa un factor constante (en este caso, normalizado con un factor 2.9).

5.3 Apartado 3

La función ordena correctamente una permutación.

5.4 Apartado 4

Tiempo



El gráfico muestra el tiempo de ejecución empírico de HeapSort (en morado) comparado con una función teórica x/60000·log(x) (en verde), que representa el comportamiento esperado de HeapSort en términos de complejidad temporal O(nlogn).

La curva empírica (en morado) sigue, en términos generales, la tendencia de la función x·log(x)/60000, lo que indica que el comportamiento experimental del algoritmo es coherente con las previsiones teóricas.

Sin embargo, se observan picos significativos en correspondencia con tamaños particulares de la entrada. Estos picos podrían deberse a factores de implementación, como asignaciones de memoria, gestión del sistema operativo o ineficiencias relacionadas con las optimizaciones del compilador.

6. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

6.1 Pregunta 1

Las trazas muestran cómo el tiempo real se alinea con la complejidad teórica, pero también presentan ligeras diferencias:

- Ambos algoritmos siguen una tendencia similar a O(nlog(n)), pero el rendimiento empírico puede variar debido a factores como eficiencia de la memoria caché, principalmente para el mergesort, el costo de las operaciones de intercambio y la optimización del compilador y hardware utilizado.
- Los picos en las gráficas pueden deberse a variaciones en Sobrecarga del sistema durante las pruebas a causa de Procesos externos y las Características del hardware, como la gestión de memoria o tiempos de acceso al disco.

6.2 Pregunta 2

En HeapSort el caso mejor es O(n log n) en todos los casos (mejor, peor y promedio), porque siempre requiere construir el heap y mantener la estructura heapificada durante las eliminaciones. Sin embargo, en términos prácticos, el mejor caso ocurre cuando el heap ya está casi ordenado y las operaciones de reestructuración son mínimas.

EnMergeSort el caso mejor es O(n log n). El proceso de dividir y combinar siempre sigue la misma lógica, independientemente del orden de entrada. Sin embargo, en la práctica, cuando las combinaciones son simples (p. ej., arreglos casi ordenados), puede ser ligeramente más rápido. También el peor y el promedio son O(n log n)

6.3 Pregunta 3

En la práctica, HeapSort puede ser ligeramente más lento que MergeSort debido a la constante multiplicativa involucrada en las operaciones de manejo del heap (como intercambios y la reestructuración del heap). Estas operaciones implican accesos más costosos en términos de memoria debido a la falta de acceso secuencial. Generalmente, MergeSort es más rápido en la práctica debido a su mejor aprovechamiento de la localidad espacial y temporal en la memoria. Al combinar subarreglos en orden secuencial, MergeSort accede a datos de manera más eficiente, lo que reduce los costos en tiempo.

Para la gestión de la memoria **HeapSort** Utiliza memoria en el lugar (O(1) espacio adicional), ya que realiza las operaciones directamente en el arreglo de entrada. Mergesort requiere memoria adicional (O(n)) para los arreglos temporales durante las combinaciones, lo que puede ser costoso para entradas grandes.

7. Conclusiones finales.

En esta práctica se implementaron y analizaron dos algoritmos de ordenamiento, MergeSort y HeapSort, validando su funcionamiento y eficiencia mediante pruebas empíricas y comparaciones con sus complejidades teóricas O(nlog⁡n)O(n \log n)O(nlogn). Los resultados obtenidos mostraron que ambos algoritmos cumplen con las predicciones teóricas, aunque con diferencias en rendimiento práctico debido a factores como el manejo de memoria y la estructura interna de cada algoritmo.

MergeSort destacó por su mejor aprovechamiento de la memoria en términos de localidad espacial y temporal, lo que le permitió una ejecución más eficiente en escenarios prácticos. Por otro lado, HeapSort, a pesar de ser menos eficiente en la práctica, demostró ventajas en términos de uso de memoria, al no requerir almacenamiento adicional significativo.

Las gráficas generadas reflejaron comportamientos consistentes con las expectativas teóricas, a excepción de algunos picos en los tiempos de ejecución, atribuidos a factores externos como la gestión del sistema operativo y características del hardware. Esto subraya la importancia de considerar tanto la teoría como los detalles prácticos al evaluar algoritmos en entornos reales.

En resumen, la práctica permitió no solo profundizar en la implementación y análisis de estos algoritmos, sino también entender las implicaciones prácticas de sus características y limitaciones, proporcionando una base sólida para su aplicación en problemas reales.