

Análisis de Algoritmos 2025/2026

Práctica 2

Rodrigo Díaz-Regañón Ureña

Daniel Martínez Fernández

Código	Gráficas	Memoria	Total

ÍNDICE

1. <i>Introducción</i>	3
2. <i>Objetivos</i>	3
2.1 Apartado 1	3
2.2 Apartado 2	4
2.3 Apartado 3	4
2.4 Apartado 4	4
2.5 Apartado 5	4
3. <i>Herramientas y metodología</i>	5
3.1 Apartado 1	5
3.2 Apartado 2	5
4. <i>Código fuente</i>	6
5. <i>Resultados, Gráficas</i>	11
6. <i>Respuesta a las preguntas teóricas</i>	18
7. <i>Conclusiones finales</i>	21

1. Introducción

En esta práctica se pretende analizar y comparar el comportamiento empírico de los algoritmos de ordenación basados en la metodología de *divide y vencerás* (DyV), en concreto los algoritmos **MergeSort** y **QuickSort**.

Ambos algoritmos se caracterizan por dividir una tabla o secuencia de datos en subconjuntos más pequeños, resolver el problema en cada subconjunto de forma recursiva y combinar posteriormente los resultados obtenidos. Esta metodología permite obtener algoritmos más eficientes que los métodos locales de ordenación (como InsertSort, SelectSort o BubbleSort), alcanzando complejidades temporales en torno a **$O(N \log N)$** en su caso medio.

El trabajo consistirá en la implementación de las versiones iterativas y recursivas de estos métodos, la medición experimental de su tiempo de ejecución y del número de operaciones básicas realizadas, y la comparación de los resultados empíricos con las estimaciones teóricas del rendimiento derivadas en el análisis del tema 2 de los apuntes.

A partir de los datos obtenidos, se elaborarán gráficas y se discutirán las diferencias entre el comportamiento teórico y el observado, identificando los casos mejor, peor y medio de cada algoritmo y analizando el impacto de la elección del pivote en QuickSort.

2. Objetivos

El objetivo general de esta práctica es estudiar experimentalmente la eficiencia de los algoritmos de ordenación recursivos de tipo divide y vencerás, verificando la validez de las cotas teóricas vistas en clase y comprendiendo los factores que influyen en su comportamiento empírico.

De forma más concreta, se busca que consigamos:

- Comprender la estructura recursiva de los algoritmos DyV.
- Aprender a instrumentar código para contar operaciones básicas.
- Comparar los tiempos experimentales con los teóricos.
- Analizar el impacto de distintas estrategias de elección del pivote en QuickSort.
- Interpretar los resultados mediante representaciones gráficas y argumentaciones razonadas.

2.1 Apartado 1

- Implementar el algoritmo MergeSort de forma recursiva, empleando las funciones mergesort y merge.
- Comprender la división de la tabla y el proceso de combinación (merge).
- Asegurar que el algoritmo ordena correctamente cualquier permutación de entrada.
- Identificar la operación básica del algoritmo (comparación de claves).

- Preparar el código para poder medir las operaciones básicas y el tiempo de ejecución.

2.2 Apartado 2

- Modificar el programa de medidas (exercise5.c) para ejecutar MergeSort sobre tablas de distintos tamaños.
- Obtener los valores del número máximo, mínimo y promedio de operaciones básicas, y los tiempos de ejecución asociados.
- Representar gráficamente los resultados y compararlos con las cotas teóricas $O(N \log N)$.
- Analizar las posibles desviaciones entre el rendimiento teórico y empírico y discutir las causas.

2.3 Apartado 3

- Implementar el algoritmo QuickSort de forma recursiva mediante la función quicksort y las rutinas auxiliares partition y median.
- Comprender el funcionamiento del proceso de partición y el papel del pivote.
- Verificar la corrección del algoritmo en distintos casos de entrada.
- Medir el número de operaciones básicas ejecutadas y el tiempo de ejecución.

2.4 Apartado 4

- Reutilizar el programa de medidas para evaluar QuickSort con distintas permutaciones y tamaños de tabla.
- Obtener las medidas de rendimiento (máximo, mínimo y promedio de operaciones básicas y tiempos).
- Comparar los resultados experimentales con las predicciones teóricas de $O(N \log N)$ en el caso medio y $O(N^2)$ en el peor caso.
- Discutir las posibles diferencias observadas con MergeSort, tanto en tiempo como en consumo de memoria.

2.5 Apartado 5

- Implementar las funciones median_avg (pivote medio) y median_stat (pivote estadístico).
- Modificar partition para emplear las nuevas estrategias de pivote.
- Evaluar y comparar empíricamente el comportamiento de las tres versiones de QuickSort (pivote primero, medio y estadístico).
- Representar las gráficas de operaciones y tiempos de ejecución.
- Discutir razonadamente los resultados, identificando qué estrategia ofrece el mejor equilibrio entre eficiencia y estabilidad.

3. Herramientas y metodología

Para la realización de esta práctica se implementaron las funciones a través de un portátil con Linux (Ubuntu) y otro MacOS, utilizando Visual Studio como entorno de programación.

Mediante la herramienta Valgrind se realizó un análisis completo para descartar cualquier tipo de problema de gestión de memoria, accesos inválidos...

3.1 Apartado 1

Se comprobó el correcto funcionamiento de la implementación de Mergesort usando Valgrind, para todo tipo de entradas en el prompt (size<0, size=0, size>0). En todos ellos la memoria se libera correctamente y no hay errores de valgrind.

3.2 Apartado 2

Se llevó a cabo un test para comprobar el funcionamiento del algoritmo implementado Mergesort haciendo varias pruebas (cambiando el prompt para el comando `exercise5_test` del makefile) y llevándolo a situaciones extremas, es decir, `num_min<0` ó `num_max<0`, `num_min>=num_max`, `incr<=0`, `numP<=0`, `outputFile` un fichero sin terminación, etc. En todos ellos la memoria se libera correctamente y no hay errores de valgrind.

3.3 Apartado 3

Se comprobó el correcto funcionamiento de la implementación de quicksort usando Valgrind, para todo tipo de entradas en el prompt (size<0, size=0, size>0). En todos ellos la memoria se libera correctamente y no hay errores de valgrind.

3.4 Apartado 4

Se llevó a cabo un test para comprobar el funcionamiento del algoritmo implementado quicksort haciendo varias pruebas (cambiando el prompt para el comando `exercise5_test` del makefile) y llevándolo a situaciones extremas, es decir, `num_min<0` ó `num_max<0`, `num_min>=num_max`, `incr<=0`, `numP<=0`, `outputFile` un fichero sin terminación, etc. En todos ellos la memoria se libera correctamente y no hay errores de valgrind.

3.5 Apartado 5

Se comprobó el correcto funcionamiento de la implementación de Quicksort con `median_avg` y `median_stat`, usando Valgrind, para todo tipo de entradas en el prompt (size<0, size=0, size>0). En todos ellos la memoria se libera correctamente y no hay errores de valgrind.

Por otro lado, se llevó a cabo un test para comprobar el funcionamiento del algoritmo implementado quicksort con `median_avg` y `median_stat`, haciendo varias pruebas (cambiando el prompt para el comando `exercise5_test` del makefile) y llevándolo a situaciones extremas, es decir, `num_min<0` ó `num_max<0`, `num_min>=num_max`,

incr<=0, numP<=0, outputFile un fichero sin terminación, etc. En todos ellos la memoria se libera correctamente y no hay errores de valgrind.

4. Código fuente

4.1 Apartado 1

```
int merge(int *tabla, int ip, int iu, int imedio)
{
    int *tabla_aux = NULL;
    int i, j, k, OB = 0;

    tabla_aux = (int *)calloc((iu - ip + 1), sizeof(int));
    if (tabla_aux == NULL)
    {
        fprintf(stderr, "Error al reservar memoria.\n");
        return ERR;
    }

    i = ip;
    j = imedio + 1;
    k = 0;

    while (i <= imedio && j <= iu)
    {
        if (tabla[i] < tabla[j])
        {
            tabla_aux[k] = tabla[i];
            i++;
        }
        else
        {
            tabla_aux[k] = tabla[j];
            j++;
        }
        k++;
        OB++;
    }

    if (i > imedio)
    {
        while (j <= iu)
        {
            tabla_aux[k] = tabla[j];
            k++;
            j++;
        }
    }
    else if (j > iu)
    {
```

```

        while (i <= imedio)
        {
            tabla_aux[k] = tabla[i];
            i++;
            k++;
        }
    }
    for (i = ip; i <= iu; i++){
        tabla[i] = tabla_aux[i - ip];
    }

    free(tabla_aux);
    return OK;
}

int Mergesort(int *array, int ip, int iu)
{
    int p_medio;
    int OB_1, OB_2, OB_3, OB;

    if (array == NULL)
    {
        fprintf(stderr, "El array es invalido.\n");
        return ERR;
    }
    if (ip < 0 || iu < 0)
    {
        fprintf(stderr, "Los indices tienen que ser positivos.");
        return ERR;
    }
    if (ip > iu)
    {
        fprintf(stderr, "El primer indice es mayor que el ultimo.\n");
        return ERR;
    }

    if (ip == iu)
    {
        return OK;
    }

    p_medio = (ip + iu) / 2;

    OB_1 = Mergesort(array, ip, p_medio);
    if (OB_1 == ERR)
        return ERR;

    OB_2 = Mergesort(array, p_medio + 1, iu);
    if (OB_2 == ERR)
        return ERR;
}

```

```

OB_3 = merge(array, ip, iu, p_medio);
if (OB_3 == ERR)
    return ERR;

OB = OB_1 + OB_2 + OB_3;
return OB;
}

```

4.3 Apartado 3

```

int partition(int *tabla, int ip, int iu, int *pos)
{
    int m, ob = 0;
    int k, aux, i;

    if (!tabla || iu < ip || !pos)
    {
        return ERR;
    }

    if (median(tabla, ip, iu, pos) == ERR)
    {
        return ERR;
    }

    m = *pos;
    k = tabla[m];
    aux = tabla[m];
    tabla[m] = tabla[ip];
    tabla[ip] = aux;
    m = ip;
    for (i = ip + 1; i <= iu; i++)
    {
        if (tabla[i] < k)
        {
            m++;
            aux = tabla[m];
            tabla[m] = tabla[i];
            tabla[i] = aux;
        }
        ob++;
    }
    aux = tabla[m];
    tabla[m] = tabla[ip];
    tabla[ip] = aux;
    *pos = m;
    return ob;
}

```

```

}

int median(int *tabla, int ip, int iu, int *pos)
{
    if (!tabla || iu < ip || !pos)
        return ERR;
    *pos = ip;
    return 0;
}

int quicksort(int *tabla, int ip, int iu)
{
    int m, ob1 = 0, ob2 = 0, ob3 = 0, ob;
    int *pos = NULL;

    if (tabla == NULL)
    {
        fprintf(stderr, "El array es invalido.\n");
        return ERR;
    }
    if (ip < 0 || iu < 0)
    {
        fprintf(stderr, "Los indices tienen que ser positivos.");
        return ERR;
    }
    if (ip > iu)
    {
        fprintf(stderr, "El primer indice es mayor que el ultimo.\n");
        return ERR;
    }

    if (ip == iu)
    {
        return 0;
    }

    if (!(pos = (int *)calloc(1, sizeof(int))))
    {
        fprintf(stderr, "Fallo reservando memoria para la variable pos.\n");
        return ERR;
    }
    ob1 = partition(tabla, ip, iu, pos);
    if (ob1 == ERR)
    {
        fprintf(stderr, "Fallo en la rutina Partir\n");
        return ERR;
    }
    m = *pos;
    if (ip < m - 1)
    {
        ob2 = quicksort(tabla, ip, m - 1);
    }
    if (iu > m + 1)
    {
        ob3 = quicksort(tabla, m + 1, iu);
    }
    free(pos);
    if (ob1 < ob2 && ob1 < ob3)
        return ob1;
    else if (ob2 < ob3)
        return ob2;
    else
        return ob3;
}

```

```

    if (ob2 == ERR)
    {
        return ERR;
    }
}
if (m + 1 < iu)
{
    ob3 = quicksort(tabla, m + 1, iu);
    if (ob3 == ERR)
    {
        return ERR;
    }
}
ob = ob1+ob2+ob3;
free(pos);
return ob;
}

```

4.5 Apartado 5

Mantenemos el mismo código para Quicksort y en partition solo cambiaremos la línea

```
ob=median(tabla, ip, iu, pos);
```

seleccionando el pivotaje que emplearemos.

Además, añadimos la implementación de los nuevos pivotajes:

```

int median_avg(int *tabla, int ip, int iu, int *pos)
{
    if (!tabla || iu < ip || !pos){
        fprintf(stderr, "Fallo en la rutina median_avg\n");
        return ERR;
    }
    *pos = (ip + iu)/2;
    return 0;
}

int median_stat(int *tabla, int ip, int iu, int *pos)
{
    int im;
    int a;
    int b;
    int c;
    int OB = 0;
    if (!tabla || iu < ip || !pos){
        fprintf(stderr, "Fallo en la rutina median_stat\n");
        return ERR;
    }
    a = tabla[ip];
    b = tabla[iu];
    if (a < b)
        im = a;
    else
        im = b;
    c = tabla[(ip + iu)/2];
    if (c < im)
        OB = 1;
    else if (c > im)
        OB = 2;
    *pos = OB;
}

```

```

}

im = (ip + iu) / 2;
a = tabla[ip];
b = tabla[im];
c = tabla[iu];

if (a < b) {
    OB++;
    if (b < c) {
        *pos = im;
    }
    else {
        OB++;
        if (a < c)
            *pos = iu;
        else
            *pos = ip;
    }
}
else {
    OB++;
    if (a < c) {
        *pos = ip;
    }
    else {
        OB++;
        if (b < c)
            *pos = iu;
        else
            *pos = im;
    }
}
return OB;
}

```

5. Resultados, Gráficas

Aquí poneis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

5.1 Apartado 1

El resultado obtenido tras realizar el test exercise4, donde se realiza la ordenación, mediante mergesort de una permutación de tamaño size, los parámetros introducidos han sido -size 20 y la salida fue la siguiente:

```

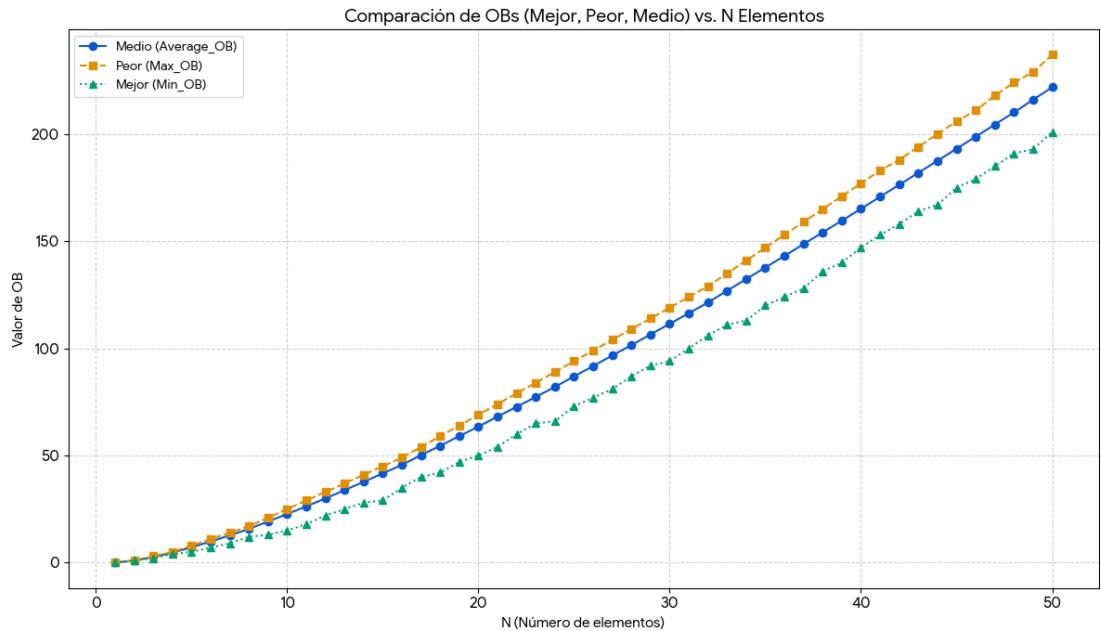
Running exercise4
Practice number 1, section 4
Done by: Rodrigo y Daniel
Group: 1272/1202
      1       2       3       4       5
      6       7       8       9       10
      11      12      13      14      15
      16      17      18      19      20

```

5.2 Apartado 2

Gráfica comparando los tiempos mejor peor y medio en OBs para MergeSort, comentarios a la gráfica.

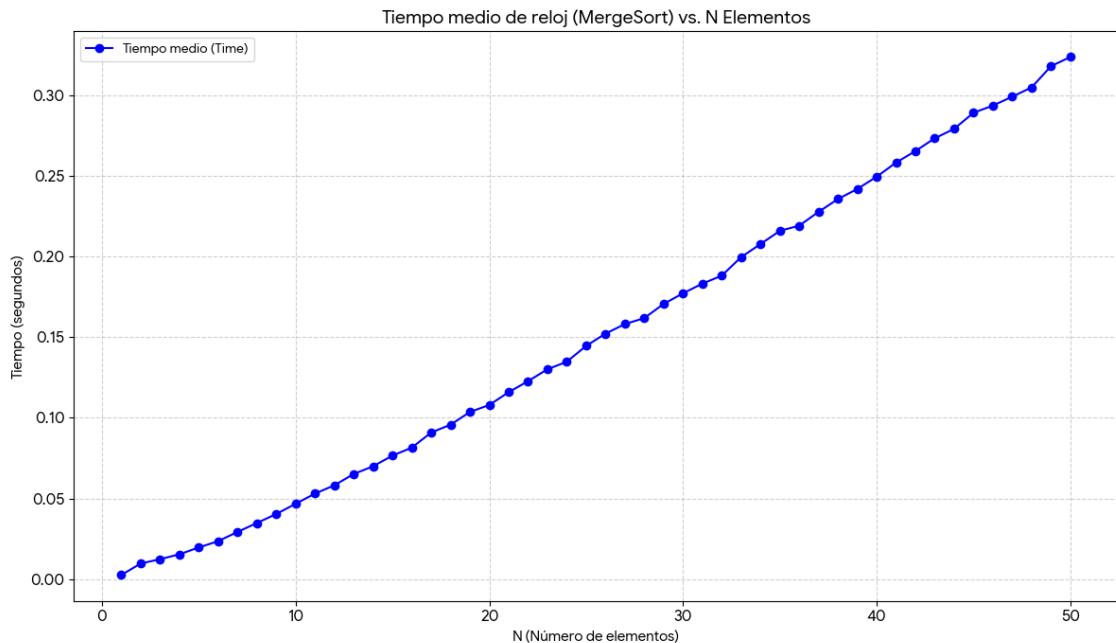
El prompt metido para obtener la tabla de datos fue: `./exercise5 -num_min 1 -num_max 50 -incr 1 -numP 100000`. Es decir que empezamos con un elemento y vamos hasta 50 elementos con un incremento de 1. El número de permutaciones que generamos es 100000, para tratar de obtener la mejor y peor permutación para cada tabla.



Se puede observar a simple vista como la gráfica cumple coherentemente que el peor caso sea el que esté por encima, el mejor el que esté por abajo y el medio entre medias de ambos.

Por otro lado, observamos como los tres siguen una función de la forma $N \log(N)$, donde el peor caso toma $N \log(N) + O(N)$, el caso medio $N \log(N)$ y el mejor caso $\frac{1}{2}N \log(N)$. Tal y como hemos visto en teoría (Los tres tienen un crecimiento de $O(N \log(N))$)

Gráfica con el tiempo medio de reloj para MergeSort, comentarios a la gráfica.



Se puede observar cómo esta gráfica sigue un crecimiento mayor que el lineal y menor que el cuadrático, que se asemeja al $N \log(N)$, por tanto, el tiempo medio en segundos concuerda con la teoría para el tiempo medio en OBs.

5.3 Apartado 3

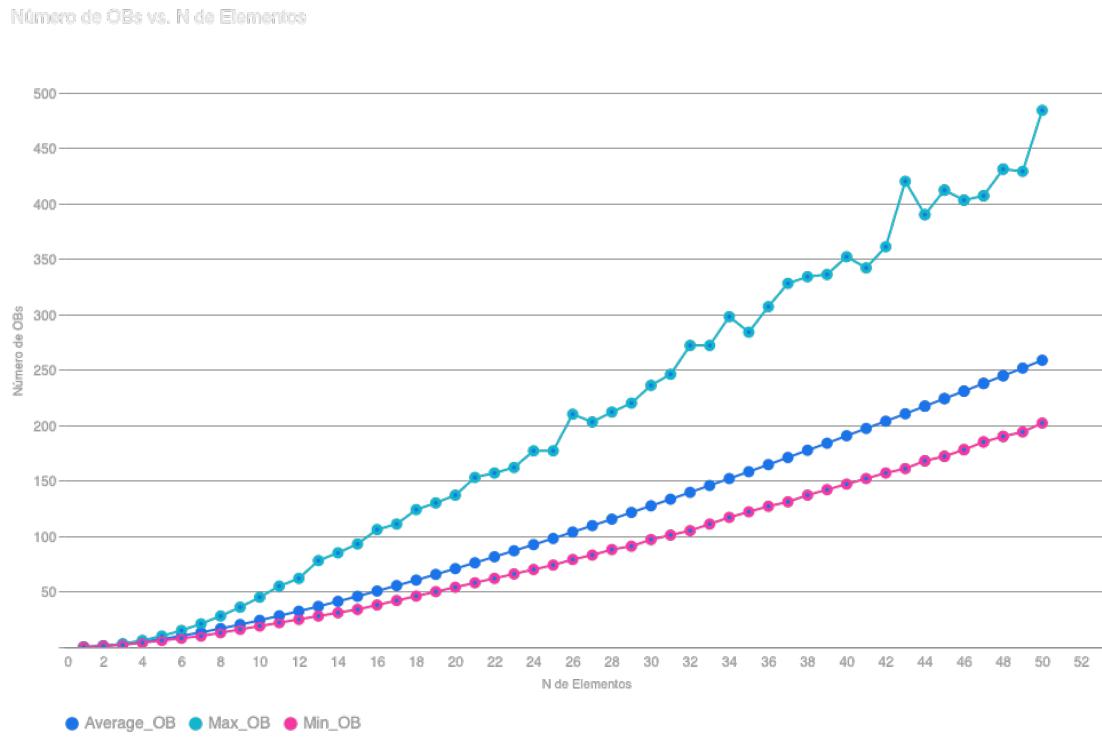
El resultado obtenido tras realizar el test exercise4, donde se realiza la ordenación, mediante mergesort de una permutación de tamaño size, los parámetros introducidos han sido -size 20 y la salida fue la siguiente:

```
Running exercise4
Practice number 1, section 4
Done by: Rodrigo y Daniel
Group: 1272/1202
1   2   3   4   5   6
    7   8   9   10  11
12  13  14  15  16
    18  19  20
```

5.4 Apartado 4

Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.

El prompt metido para obtener la tabla de datos fue: `./exercise5 -num_min 1 -num_max 50 -incr 1 -numP 100000`. Es decir que empezamos con un elemento y vamos hasta 50 elementos con un incremento de 1. El número de permutaciones que generamos es 100000, para tratar de obtener la mejor y peor permutación para cada tabla.



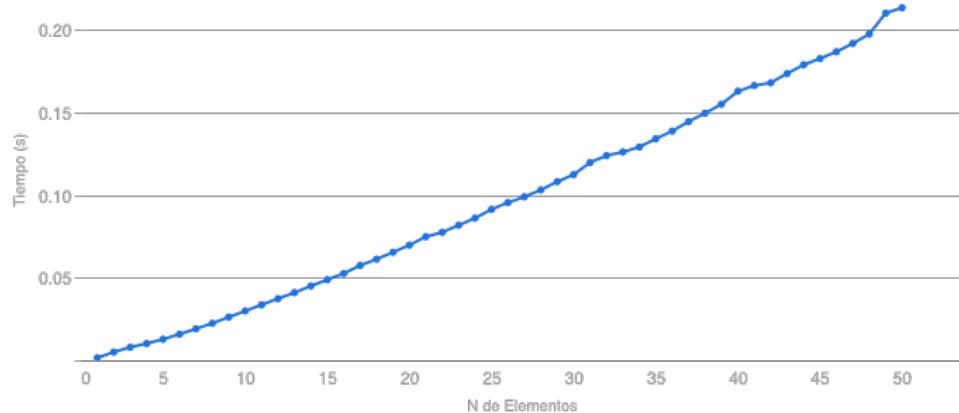
Como podemos observar la gráfica se adapta perfectamente a lo estudiado en teoría, el caso mejor, pues sigue un crecimiento logarítmico $B_{QS}(N) = N \log(N) + O(N)$

En el caso medio se puede intuir un crecimiento similar al logarítmico, que concuerda con lo visto en teoría: $A_{QS}(N) = 2N \log(N) + O(N)$

En el caso peor se pueden observar algunos picos a medida que va creciendo el número de elementos. Vemos que las primeras perturbaciones se encuentran a partir de $N = 10$, esto se debe a que el número de permutaciones para 10 elementos es $10! = 3628800$, lo cual supera con creces las 100000 permutaciones que generamos así que probabilísticamente es muy complicado que se genere justo la peor permutación $(1, 2, \dots, N)$, de 10 en adelante, esto influye haciendo que adopte un crecimiento más parecido al del promedio. No obstante, podemos apreciar como sigue un crecimiento cuadrático, que concuerda con la teoría $W_{QS}(N) = \frac{N^2}{2} - \frac{N}{2}$.

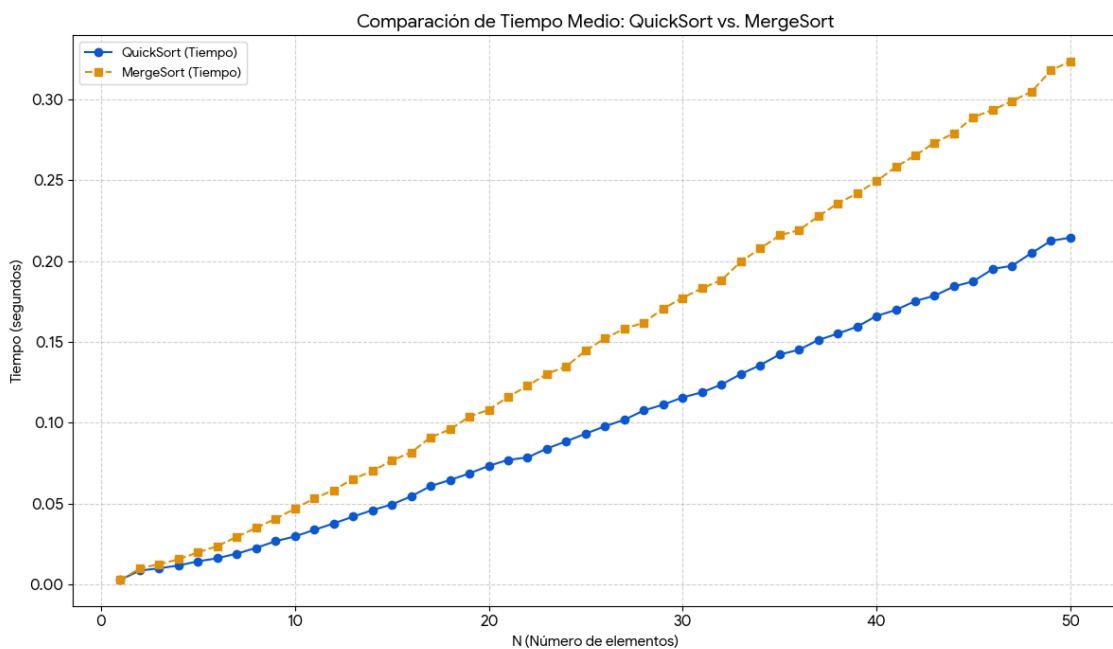
Gráfica con el tiempo medio de reloj para QuickSort, comentarios a la gráfica.

Tiempo de Ejecución vs. N de Elementos



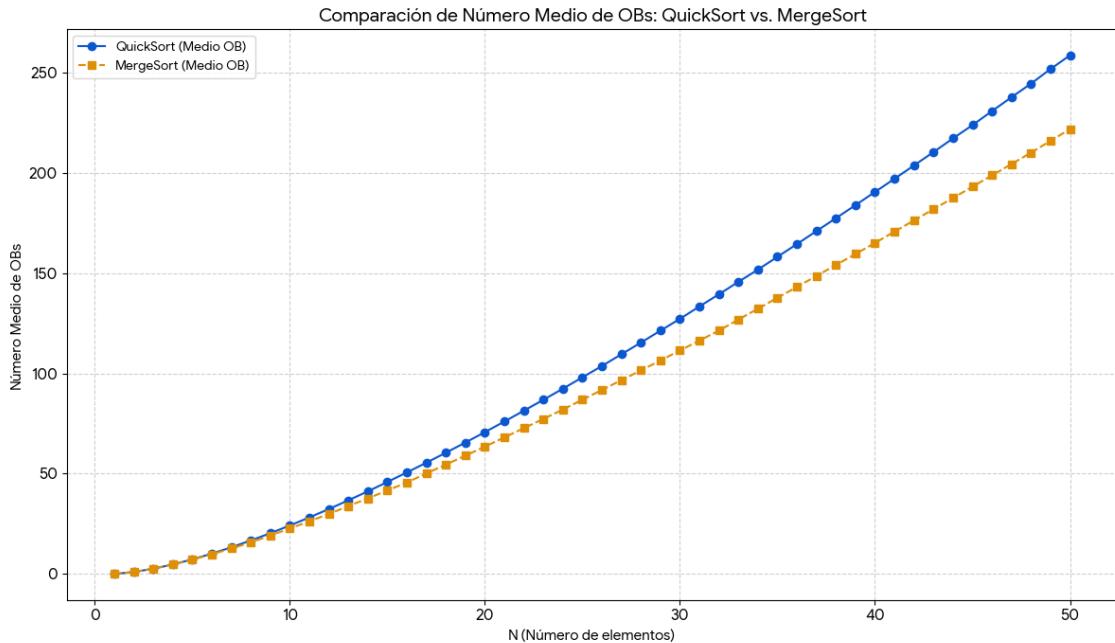
Podemos observar que esta gráfica no sigue un crecimiento cuadrático ni lineal, se encuentra en un intermedio, que corresponde a $O(n\log n)$, lo que tiene sentido cuando se compara con la teoría.

Grafica comparando el tiempo medio de reloj de MergeSort y QuickSort



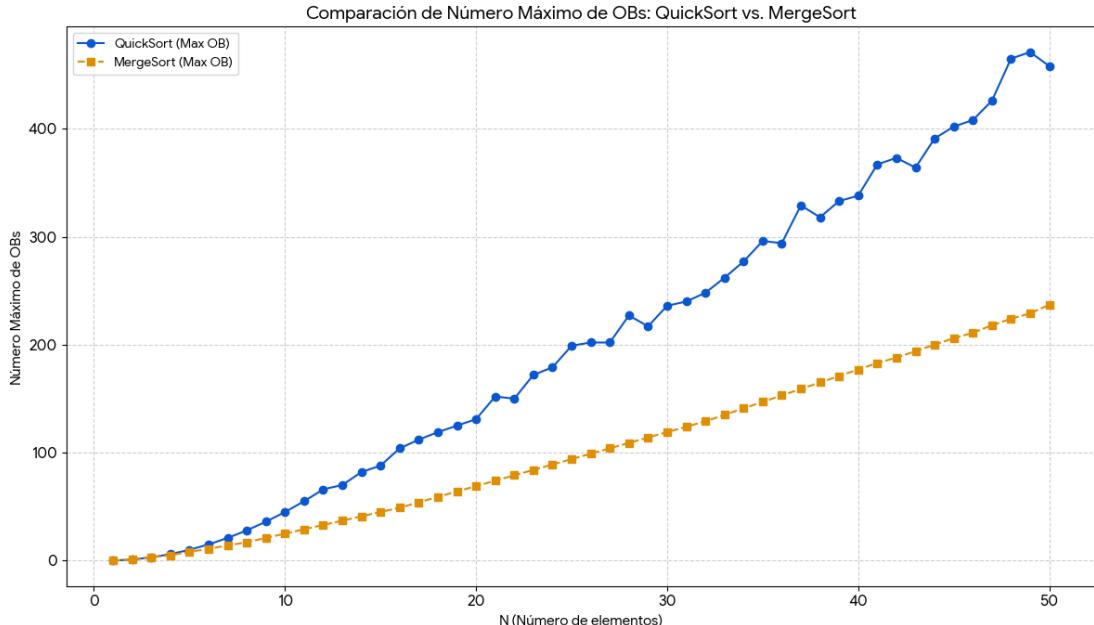
Observamos como para toda tabla Quicksort, es mejor que mergesort, pues como veremos más adelante tiene su sentido. Por ello se puede decir que por lo general, Quicksort es mejor para Mergesort, no obstante, veremos que mergesort tiene sus ventajas

Grafica comparando el número medio de OBs de MergeSort y QuickSort



Observamos en esta gráfica cómo el tiempo medio en OBs en Quicksort es notablemente mejor al de Mergesort, esto concuerda con la teoría pues $A_{QS}(N) = 2N\log(N) + O(N)$ y $A_{MS}(N) = N\log(N)$, por lo tanto, Quicksort crece $2 * A_{MS}(N) + O(N)$. De esta forma, el apartado anterior cobra más sentido

Grafica comparando el número máximo de OBs de MergeSort y QuickSort



Como antes explicamos, el caso peor no es regular para Quicksort, no obstante, vemos que aun no cogiendo el verdadero caso peor, realiza muchas más operaciones básicas que en el caso peor de mergesort. El caso peor de mergesort si es continuo, ya que como

hemos visto en la teoría todas las permutaciones tienen el mismo número de comparaciones (todas son el caso peor), esto hace que aunque el caso peor sea mejor que el de Quicksort, el caso medio es peor como vimos antes.

5.5 Apartado 5

El resultado obtenido tras realizar el test exercise4, donde se realiza la ordenación, mediante Quicksort, usando median_avg, de una permutación de tamaño size. Los parámetros introducidos han sido -size 20 y la salida fue la siguiente:

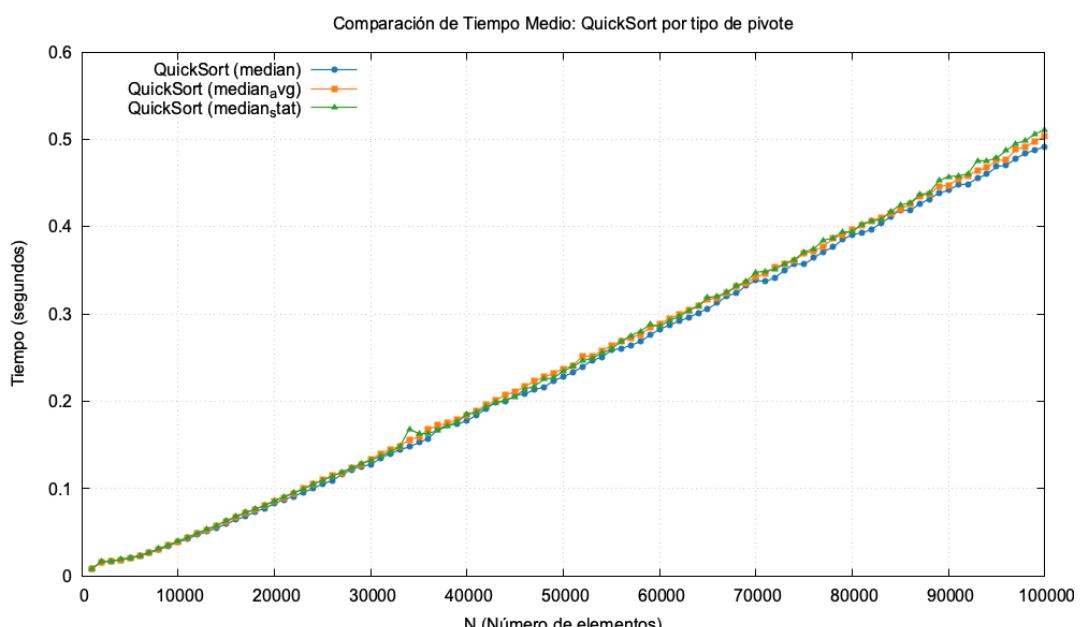
```
Running exercise4
Practice number 2, section 5.1
Done by: Rodrigo y Daniel
Group: 1272/1202
1   2   3   4   5   6
7   8   9   10  11  12  13  14  15  16  17  18
19  20
```

El resultado obtenido tras realizar el test exercise4, donde se realiza la ordenación, mediante Quicksort, usando median_stat, de una permutación de tamaño size. Los parámetros introducidos han sido -size 20 y la salida fue la siguiente:

```
Practice number 2, section 5.2
Done by: Rodrigo y Daniel
Group: 1272/1202
1   2   3   4   5   6
7   8   9   10  11  12  13  14  15  16  17  18
19  20
```

Gráfica comparando el tiempo medio de reloj de Quicksort usando los pivotes median, median_avg y median_stat.

El prompt insertado para obtener la tabla de datos fue: ./exercise5 -num_min 1000 -num_max 100000 -incr 1000 -numP 50. Es decir que empezamos con 1000 elementos y vamos hasta 100000 elementos con un incremento de 1000. El número de permutaciones que generamos es 50, de forma que buscamos conseguir una media razonable para cada N.



Podemos ver que los tres son muy similares, y siguen el crecimiento teórico de Quicksort, es decir $O(N \log N)$.

Observamos que, durante la tabla, generalmente Quicksort con median se mantiene por debajo del resto y medianavg y medianstat se van alternando, aunque por lo general medianstat se mantiene por encima. Estas discontinuidades se deben a que estamos usando tablas de muchísimos elementos y las muestras cogidas son sólo de 50 permutaciones, por tanto puede ser que a veces una sea mejor y otra peor.

No obstante, observando la tabla podemos intuir que median es la forma más rápida, medianavg es intermedia y medianstat es la que más tarde, aunque para demostrar esto habría que hacer $N!$ permutaciones para cada N y que el sistema hiciese las operaciones con la misma velocidad todo el rato, cosa que es imposible.

6. Respuesta a las preguntas teóricas.

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

6.1 Pregunta 1

Hemos ido tratando esta pregunta en los comentarios a las gráficas pero haremos un repaso.

En la gráfica de comparación del tiempo medio entre QuickSort y MergeSort se observa que ambas curvas presentan un crecimiento suave y regular, sin picos ni irregularidades pronunciadas. Esto indica que el rendimiento empírico en las pruebas se ha mantenido estable y cercano al comportamiento medio teórico previsto para cada algoritmo.

En particular:

- **MergeSort** presenta un crecimiento de tipo $O(N \log(N))$ tanto teórico como empírico (hicimos el razonamiento basándonos en que está entre función lineal y cuadrática).
- **QuickSort**, en el caso medio, también se ajusta bien a una función del tipo $2N \log(N) + O(N)$, es decir, sigue un crecimiento de $O(N \log(N))$, por el razonamiento anterior, aunque dependiendo del pivote elegido puede presentar mayor o menor dispersión, en el caso de median se muestra más o menos regular.

En conclusión, las curvas empíricas obtenidas coinciden con el comportamiento teórico $O(N \log(N))$, y la suavidad de las gráficas confirma que el algoritmo se ejecuta en su caso medio sin anomalías debidas al pivote.

6.2 Pregunta 2

Esta pregunta se ha respondido en el apartado 5.5. Haciendo un resumen, no es posible con nuestros medios determinar cuál es el mejor pivote para usar al 100%, no obstante, podemos intuir, que el medianstat es el que más tarda, y el median el que menos.

6.3 Pregunta 3

MergeSort siempre divide el array en dos mitades, el coste de operación viene a la hora de combinar las tablas, si cuando combinamos, todos los elementos de una tabla son menores que los de otra, entonces tenemos el caso mejor [1 2 3 4 5 6 7]. Si por el contrario en todas las combinaciones llegamos al punto en el que queda un elemento en cada subtabla que se está combinando, estamos ante el caso peor [7 3 1 6 2 5 4]. No obstante, ambas tienen coste ($O(N \log N)$) Como vimos en el apartado 5.2

QuickSort con median tiene mejor caso cuando el pivote divide el array en particiones equilibradas ($O(N \log N)$) [4 2 6 1 3 5 7] y peor caso cuando el pivote queda siempre en un extremo de un array ordenado o inversamente ordenado ($O(N^2)$) [1 2 3 4 5 6 7].

QuickSort con median_avg tiene mejor caso cuando las particiones quedan equilibradas ($O(N \log N)$) [1 2 3 4 5 6 7] y peor caso cuando, por azar o diseño adverso, las particiones quedan muy desbalanceadas ($O(N^2)$) [7 6 5 4 3 2 1].

QuickSort con median_stat tiene mejor caso cuando las particiones quedan equilibradas ($O(N \log N)$) [3 6 1 4 2 7 5] y peor caso cuando, por azar o diseño adverso, las particiones quedan muy desbalanceadas ($O(N^2)$) [1 2 3 4 5 6 7], aunque la probabilidad de este último caso es baja en la práctica.

Para generar el mejor y peor caso forzadamente en la práctica, podríamos modificarlo creándonos una función que genere una permutación que nos dé el mejor y peor caso para cada algoritmo, en merge sort podríamos crear la permutación mejor con bucle for:

```
for (int i = 0; i < N; i++)
```

```
    tabla[i] = i + 1;
```

Y este serviría para el peor caso de Quicksort median y para el mejor de median_avg.

Y como peor caso de merge la clave es ir dividiendo la tabla en los índices impares y pares y hacerlo recursivamente hasta que nos queden tablas de 2 elementos, y entonces las vamos ordenando, un ejemplo para N=7 sería:

Comienza con [1, 2, 3, 4, 5, 6, 7]

Divide en posiciones pares e impares:

Pares → [1, 3, 5, 7]

Impares → [2, 4, 6]

Aplica el mismo proceso recursivamente:

$[1, 3, 5, 7] \rightarrow [1, 5] + [3, 7] \rightarrow [1, 5, 3, 7]$

$[2, 4, 6] \rightarrow [2, 6] + [4] \rightarrow [2, 6, 4]$

Combina: $[1 \ 5 \ 3 \ 7 \ 2 \ 6 \ 4]$.

Esto se podría programar como una función con entrada N.

Por último:

Para median el mejor caso será, que el elemento $\frac{N}{2} + 1$ esté en primer lugar, para cada subtabla.

Y para median_avg y median_stat el peor caso será, que la tabla esté ordenada de manera descendente.

Para generar el mejor caso de QuickSort recursivamente se elige como pivote la mediana del array, se coloca en la posición actual, y se aplica recursivamente el mismo proceso a las subtablas de elementos menores y mayores.

Esto se podría formalizar en un programa en c para tablas de N elementos.

6.4 Pregunta 4

El algoritmo empíricamente más eficiente es Quicksort, el razonamiento se puede ver en el apartado 5.4 (tercera gráfica). Esta conclusión no concuerda con lo estudiado en teoría (se observa en la cuarta gráfica del 5.4). Pero tiene sentido, ya que aunque a nivel de Obs Quicksort sea más eficiente, la elección del pivote y los intercambios de elementos añaden tiempo de ejecución, mientras que en mergesort, no se hace ningún intercambio ni elección del pivote, simplemente se divide la tabla para luego combinarla (por lo que la única operación básica de peso es la comparación, mientras que en Quicksort lo puede ser el intercambio de elementos).

De este modo, vemos que Mergesort será más eficiente que cualquier forma de Quicksort a nivel empírico.

A nivel de memoria, sin embargo, será más eficiente Quicksort, pues sólo reservamos memoria para `*pos`, que es un entero, mientras que para Mergesort estaremos reservando memoria para tablas auxiliares de muchos elementos enteros.

7. Conclusiones finales.

En esta práctica hemos analizado de forma experimental el comportamiento de los algoritmos de ordenación recursivos MergeSort y QuickSort (con distintas estrategias de pivote: median, median_avg y median_stat), comparando sus tiempos de ejecución, número de operaciones básicas y uso de memoria con las predicciones teóricas.

Los resultados obtenidos confirman que MergeSort es, en términos empíricos, más rápido que QuickSort para la mayoría de los tamaños de tabla probados.

Además, MergeSort presenta un comportamiento más regular y predecible, con tiempos de ejecución y número de operaciones que siguen de manera estricta la complejidad ($O(N \log N)$), independientemente de la permutación de entrada. Esto lo hace más estable y confiable para casos en los que se requiere un rendimiento garantizado.

En cuanto a la gestión de memoria, QuickSort es claramente más eficiente, ya que sólo necesita memoria adicional para la variable auxiliar de la posición del pivote y la pila de llamadas recursivas, mientras que MergeSort requiere reservar arrays auxiliares de tamaño proporcional a la tabla, lo que puede ser relevante para tablas grandes.

Respecto a las estrategias de pivote en QuickSort, los experimentos muestran que median tiende a ser la opción más rápida, seguida de median_avg, mientras que median_stat suele ser ligeramente más lenta debido a la sobrecarga en la selección estadística del pivote. No obstante, la diferencia es pequeña y, en la práctica, las tres variantes siguen el crecimiento teórico ($O(N \log N)$) en el caso medio.

Por último, la práctica permite concluir que la elección del algoritmo y la estrategia de pivote depende del tipo de entrada, tamaño del array y restricciones de memoria. MergeSort es recomendable cuando se requiere estabilidad y previsibilidad, mientras que QuickSort es más adecuado cuando se busca velocidad y eficiencia en memoria, especialmente para entradas grandes y aleatorias.

En conjunto, la experiencia empírica coincide en gran medida con la teoría, validando los análisis de complejidad vistos en clase, y permite apreciar de manera clara el impacto que la estructura de los datos y la implementación tienen sobre el rendimiento de los algoritmos de ordenación DyV.