Ficha 31

Estrategias de Resolución de Problemas: Divide y Vencerás

1.] Aplicaciones de la recursividad: El algoritmo Quicksort.

Si se observa el método de *Intercambio Directo* o *Bubblesort*, se verá que algunos elementos (los mayores o "más pesados") tienden a viajar más rápidamente que otros (los menores o "más livianos") hacia su posición final en el arreglo. El proceso de "burbujeo" que lleva a cabo ese algoritmo directo es evidentemente *asimétrico*, pues los elementos más grandes se comparan más veces que los más chicos en la misma pasada, haciendo que en esa misma pasada cambien de lugar también más veces que los valores pequeños. Se suele llamar *liebres* y *tortugas* a estos elementos, en obvia alusión a su velocidad de traslado por el vector [5]. El resultado es que el algoritmo necesita muchas más pasadas hasta que se acomode el último de los menores que viaja desde la derecha...

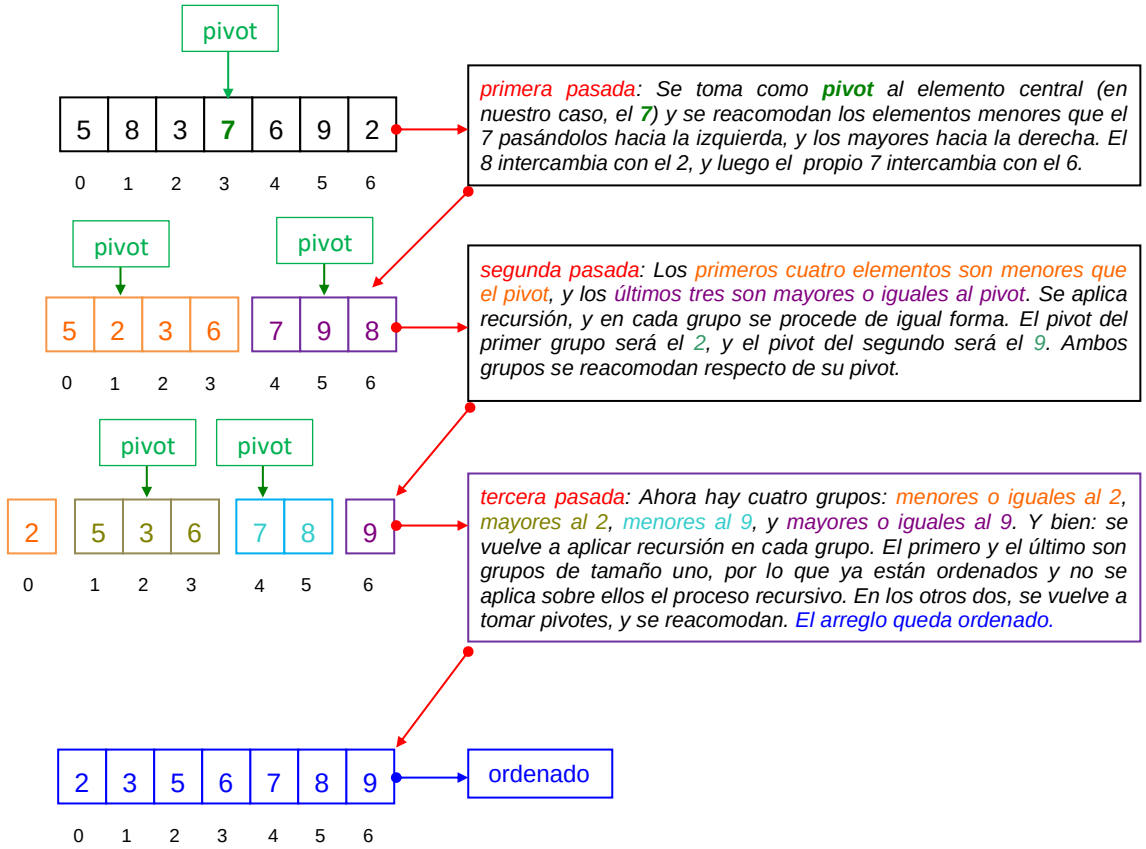
En 1960, un estudiante de ciencias de la computación llamado *Charles Hoare* se haría famoso al presentar en *Communicatons of the ACM* una versión mejorada del *bubble sort*, al cual llamó simplemente *Ordenamiento Rápido* o *Quicksort*... y desde entonces ese método se ha convertido en el más estudiado de la historia de la programación, entre otras cosas por ser hasta ahora en promedio el algoritmo más rápido (aunque hay situaciones en que se comporta bastante mal, esas situaciones de peor caso son raras y poco probables y de todos modos se puede refinar el algoritmo original para que "se proteja" de esos casos y mantenga su buen rendimiento) [6] [1].

La idea es recorrer el arreglo desde los dos extremos. Se toma un elemento *pivot* (que suele ser el valor ubicado al medio del arreglo pero puede ser cualquier otro). Luego, se recorre el arreglo desde la izquierda buscando algún valor que sea *mayor que el pivot*. Al encontrarlo, se comienza una búsqueda similar pero desde la derecha, ahora buscando un *valor menor al pivot*. Cuando ambos hayan sido encontrados, se intercambian entre ellos y se sigue buscando otro par de valores en forma similar, hasta que ambas secuencias de búsqueda se crucen entre ellas. De esta forma, se favorece que tanto los valores mayores ubicados muy a la izquierda como los menores ubicados muy a la derecha, viajen rápido hacia el otro extremo... y todos se vuelven liebres.

Al terminar esta pasada, se puede ver que el arreglo no queda necesariamente ordenado, pero queda claramente *dividido en dos subarreglos*: *el de la izquierda contiene elementos que son todos menores o iguales al pivot*, y *el de la derecha contiene elementos mayores o iguales al pivot*. Por lo tanto, ahora se puede aplicar exactamente el mismo proceso a cada subarreglo, usando *recursividad*. El mismo método que particionó en dos el arreglo original,

se invoca a sí mismo dos veces más[1], para partir en otros subarreglos a los dos que se obtuvieron recién. Con esto se generan cuatro subarreglos, y con más recursión se procede igual con ellos, hasta que sólo se obtengan particiones de tamaño uno. En ese momento, el arreglo quedará ordenado. El algoritmo procede como se ve en la *Figura 1*:

Figura 1: Esquema general de funcionamiento del algoritmo *Quicksort*.



En el proyecto *[F31] Divide y Venceras* que viene con esta Ficha, el módulo *ordenamiento.py* incluye la siguiente función *quicksort()* que implementa el algoritmo (de hecho, el módulo *ordenamiento.py* es básicamente el mismo que ya venía con la Ficha en la que se presentó el tema del Ordenamiento) [5]:

```
def quick_sort(v):
    # ordenamiento Quick Sort
    quick(v, 0, len(v) - 1)


def quick(v, izq, der):
    x = get_pivot(v, izq, der)

    i, j = izq, der
```

[1] La idea de un concepto o un proceso que recurre a sí mismo para completarse puede aplicarse a la extraña situación de un sueño dentro de un sueño, que fue utilizada ya por *Edgar Alan Poe* en 1849 en su poema "*A Dream Within a Dream*". Este poema a su vez, inspiró en 1976 una composición musical instrumental de la banda *The Alan Parsons Project* tambien llamada "*A Dream Within a Dream*". Y en el cine, la referencia es obvia: la película *Inception* (conocida como *El Origen* en Hispanoamérica) del año 2010, dirigida por *Christopher Nolan* y protagonizada por *Leonardo DiCaprio*, narra una atrapante historia de un equipo de espías industriales que navega en ¡tres niveles de sueños recursivos!

```
        while i <= j:
            while v[i] < x and i < der:
                i += 1
            while x < v[j] and j > izq:
                j -= 1
            if i <= j:
                v[i], v[j] = v[j], v[i]
                i += 1
                j -= 1

        if izq < j:
            quick(v, izq, j)

        if i < der:
            quick(v, i, der)


    def get_pivot(v, izq, der):
        # calculo del pivot: elemento central de la partición [izq, der]
        x = v[int((izq + der) / 2)]
        return x
```

El mismo proyecto contiene el programa *main.py*, que a través de un menú de opciones permite crear y ordenar un arreglo seleccionando el algoritmo de ordenamiento a aplicar (incluyendo al *Quicksort* que acabamos de mostrar). Otra vez, el programa *main.py* es esencialmente el mismo que ya se presentó con la *Ficha 17*.

2.] La estrategia Divide y Vencerás para resolución de problemas.

El algoritmo *Quicksort* que hemos analizado en la sección anterior es un ejemplo de aplicación de una *estrategia de resolución de problemas* muy conocida y estudiada, que se designa como estrategia *Divide y Vencerás*, por la cual los *n* datos del problema estudiado se dividen en subconjuntos de aproximadamente el mismo tamaño (*n/2* por ejemplo), luego se procesa cada uno de esos subconjuntos en *forma recursiva* y finalmente se "unen las partes" que se acaban de procesar para lograr el resultado final. Muchas veces, y dependiendo de factores tales como el *tamaño de cada subconjunto* o la *cantidad de invocaciones recursivas* que se hagan para procesar esos subconjuntos, esta estrategia favorece el diseño de algoritmos muy eficientes en cuanto al tiempo de ejecución [6].

Como toda estrategia general de resolución de problemas, la técnica *divide y vencerás* puede aplicarse en ciertos problemas (cuya estructura interna admita la división en subconjuntos de tamaños similares) para lograr soluciones con mejor o mucho mejor tiempo de ejecución en comparación a soluciones intuitivas de *fuerza bruta*, que suelen consistir en explorar todas y cada una de las posibles combinaciones de procesamiento de los datos de entrada, pero de tal forma que la cantidad de operaciones a realizar aumenta de manera dramática a medida que crece la cantidad *n* de datos [1].

En el algoritmo *Quicksort* la estrategia *Divide y Vencerás* es clara: se trata de dividir el arreglo en dos mitades tales que una de ellas contenga elementos menores al pivot y la otra contenga elementos mayores a ese pivot. Mediante recursión cada uno de esos subarreglos se vuelve a dividir, aplicando en cada uno el mismo proceso de intercambio con respecto al pivot. Finalmente, cuando ya no sea posible volver a particionar por haber llegado a subarreglos de tamaño 1, terminar el proceso dejando los subarreglos ordenados de forma que el arreglo final quede ordenado a su vez.

La división del arreglo en dos para volver a tomar pivotes en cada subarreglo, puede hacerse recursivamente: en cada mitad se calcula un nuevo pivot, se vuelven a trasladar los menores y los mayores; y recursivamente se vuelve a dividir, una y otra vez, obteniendo particiones de tamaños *n/2*, *n/4*, *n/8*, etc., aplicando recursión sobre ellas hasta que la partición a procesar tenga un solo elemento.

¿Qué tan bueno es este algoritmo en tiempo de ejecución? Podemos ver que la función *quick()* comienza haciendo primero el cálculo del pivot y el traslado de los menores y mayores con respecto a ese pivot. Si se analiza con cuidado ese proceso, puede verse que se ejecuta en tiempo lineal (*O(n)*): los dos ciclos más internos mueven los índices *i* y *j* una vez por cada iteración, y aun cuando ambos están dentro de otro ciclo, este ciclo externo solo controla que *i* y *j* no se crucen, pero ***no fuerza*** a los ciclos internos a comenzar nuevamente desde cero (lo que provocaría un rendimiento cuadrático).

Luego de terminar el *proceso de pivoteo*, la función *quick()* hace *dos llamadas recursivas* tomando cada una el subarreglo que corresponda (el primero, delimitado por los índices *izq* y *j*, y el segundo delimitado por *i* y *der*). Para analizar el comportamiento de la función *quick()* en cuanto a su tiempo de ejecución total, veámosla en un esquema de pseudocódigo simplificado:

```
quick(subarreglo actual):
    x = pivot del subarreglo actual
    pivotear: trasladar menores y mayores a x [==> tiempo adicional: O(n)]
    quick(mitad izquierda)
    quick(mitad derecha)
```

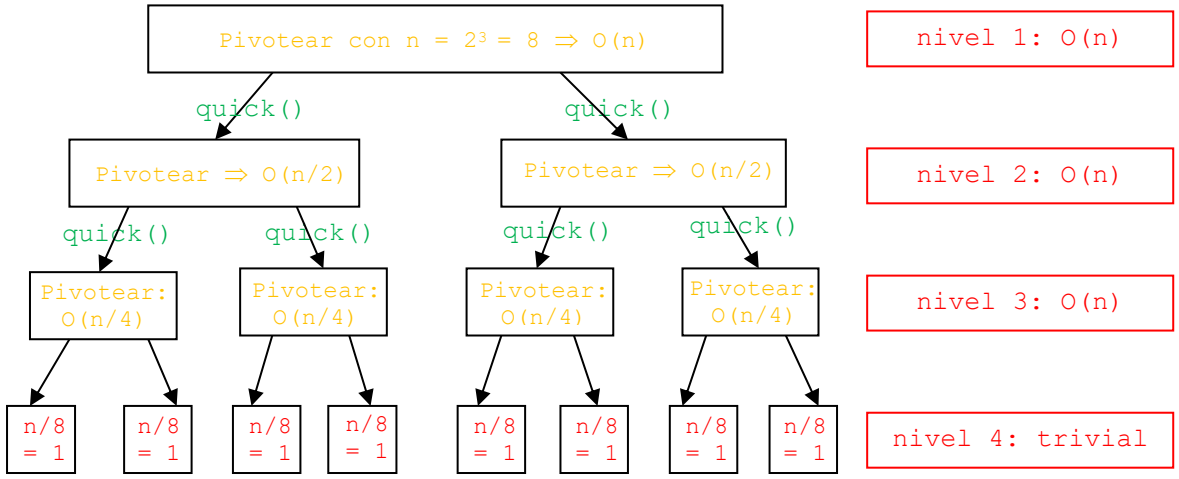
Podemos ver que el proceso completo consta de dos llamadas recursivas (cuyo tiempo de ejecución o costo ignoramos ahora) más la ejecución del proceso *pivotear* que en este caso sabemos que es de *tiempo lineal (O(n))*. Esto significa que el tiempo total empleado por la función *quick() completa*, dependerá del tiempo que lleve la ejecución de las dos llamadas recursivas. Necesitamos entonces averiguar cuántas invocaciones recursivas serán realizadas y cuánto tiempo empleará cada una.

Para ello ayudará un ejemplo. Supongamos que la cantidad de elementos del arreglo es *n = 8* (para facilitar la explicación suponemos que *n* es una potencia de 2, pero el análisis no cambia para cualquier otro valor). Al ser invocada por primera vez, la función *quick()* hace el primer proceso de pivoteo sobre el arreglo completo (tiempo: *O(n)* para cumplir este paso), y luego dos llamadas recursivas (*nivel 1* del gráfico siguiente)

Cada una de las dos instancias recursivas tiene que volver a aplicar el proceso de pivoteo sobre dos subarreglos de tamaño promedio *n/2 = 4*. En cada uno demora *O(n/2)*, por lo que el tiempo conjunto de ambos lleva a O(*n/2) + O(n/2) = O(n)* en el *nivel 2* del gráfico.

Luego en ese mismo nivel 2 se hacen cuatro llamadas recursivas a *quick()* para otros cuatro subarreglos de tamaño promedio *n/4,* por lo cual el proceso de pivoteo completo para ese nivel es *O(n/4) +O(n/4) + O(n/4) + O(n/4) = O(n)* también en el *nivel 3*. En cada nivel de llamadas recursivas, el tamaño promedio de cada subarreglo se divide por 2, y aunque es cierto que el proceso de pivoteo trabaja entonces cada vez menos, el hecho es que ese pivoteo se hace varias veces por nivel, con un tiempo de ejecución combinado que siempre es *O(n)*. ¿Cuántos *niveles* de llamadas a *quick()* tiene el árbol? La respuesta es: $k = 3 = \log_2(8) = \log_2(n)$ [o sea: tantos como se pueda dividir sucesivamente a *n* por 2 hasta llegar a un cociente de 1 (lo que finalmente es igual a $\log_2(n)$)]. En notación *Big O* se puede prescindir de la base del logaritmo, por lo que la cantidad de niveles con llamadas a *quick()* es *O(log(n))*.

Figura 2: La estrategia *Divide y Vencerás* aplicada en el algoritmo *Quicksort*.



En la gráfica anterior se muestra el nivel 4 del árbol, aunque en realidad ese nivel no llega a generarse: cuando un subarreglo tiene tamaño 1, no hay invocación recursiva para él ya que teniendo un único componente está ya ordenado. Por lo tanto, si la cantidad total de niveles de invocación a *quick()* es *O(log(n))* y cada nivel se procesa en un tiempo de ejecución combinado *O(n)* entonces el tiempo total de ejecución es *O(n*log(n))* lo cual es *subcuadrático*: puede verse que $n*log(n) < n*n = n^2$ por lo que $O(n*log(n)) < O(n^2)$. Esto prueba que *en promedio*, el algoritmo *Quicksort* ejecuta en tiempo *O(n*log(n))* (decimos *en promedio*, porque el tamaño de cada subarreglo no es constantemente igual a *n/2* en cada nivel, pero si los datos vienen aleatoriamente dispuestos se puede esperar que los subarreglos tengan tamaños que *en promedio* se asemejen a *n/2*, compensando las diferencias).

¿Qué tan general es este resultado? La estrategia *divide y vencerás* se aplicó para diseñar un algoritmo de ordenamiento y al analizarlo hemos obtenido un tiempo de ejecución promedio de *O(n*log(n)* ¿Puede inferirse que siempre será así? ¿Toda vez que se aplique la estrategia *divide y vencerás* se obtendrá una solución con tiempo de ejecución *O(n*log(n))* para el caso promedio o para el peor caso? Se puede probar que la respuesta es ***sí, siempre y cuando el algoritmo analizado tenga la misma estructura que el algoritmo quick()***: dos llamadas recursivas aplicadas sobre lotes de datos de tamaño aproximadamente igual a la mitad del lote actual, más *un proceso adicional de tiempo de ejecución lineal (O(n))* (como nuestro proceso de *pivoteo*):

Figura 3: Esquema esencial de un algoritmo *Divide y Vencerás* con tiempo de ejecución *O(n*log(n))*.

```
proceso(partición):
      adicional()  [⇒ O(n)]
      proceso(partición/2)
      proceso(partición/2)
```

⇒ O(n*log(n))

O bien:

```
proceso(partición):
      proceso(partición/2)
      proceso(partición/2)
      adicional()  [⇒ O(n)]
```

⇒ O(n*log(n))

3.] Implementación del Quicksort mediante selección del pivot por *Mediana de Tres*.

Hemos visto que el algoritmo *Quicksort* ejecuta con tiempo promedio de *O(n*log(n))*. Aún cuando ese comportamiento es muy bueno, existen casos de configuraciones de entrada que pueden llevar al *Quicksort* a un *peor caso* muy malo en cuanto a tiempo de ejecución, dependiendo de la forma en que se tome el pivot: puede probarse que si los elementos del arreglo están distribuidos de forma tal que cada vez que se toma el pivot su valor es el *menor* (o el *mayor*) en el subarreglo considerado, entonces *Quicksort* ejecuta en tiempo $O(n^2)$ (*cuadrático…*) [1].

Intuitivamente puede verse que esto es cierto: si recurrentemente se toma el *menor* o el *mayor* del subarreglo a pivotear, entonces se producirán dos particiones pero una de ellas tendrá un solo elemento y la otra tendrá *n-1* elementos… y al repetirse este patrón, el árbol de llamadas recursivas tendrá *n* niveles en lugar de *log(n)* niveles, por lo que surge el orden cuadrático: si cada nivel se pivotea a un costo de *O(n)*, y tenemos *O(n)* niveles, entonces el tiempo total combinado será $O(n^2)$.

Las distribuciones de datos que pueden causar que *Quicksort* se degrade a un rendimiento cuadrático dependen de la forma en que se selecciona el pivot. Algunos programadores suelen elegir como pivot al *primer elemento del subarreglo analizado* (o bien suelen elegir el *último*). Sin embargo, estas dos formas simples de selección del pivot son justamente las que *maximizan la posibilidad de caer en un rendimiento cuadrático*: si el arreglo estuviese ya ordenado, el algoritmo demoraría un tiempo de orden cuadrático en no hacer nada más que comparaciones, sin intercambio alguno. Por lo tanto, es una muy mala idea tomar como pivot a alguno de los elementos de los extremos en cada partición [6] [1].

La variante que hemos mostrado implementada en las secciones anteriores de esta Ficha, toma como pivot al *elemento central del subarreglo analizado*, y esta es una estrategia muy estable incluso si el arreglo ya estuviese ordenado (en este caso, se invierte un tiempo de *O(n*log(n))* en recorrer el arreglo, sin hacer intercambios). Sin embargo, aún puede ocurrir que aparezca alguna distribución de datos en la cual el elemento central elegido siempre sea el menor o el mayor. Hemos dicho que esa distribución es altamente improbable, pero aún así muestra que la estrategia de selección del pivot podría mejorarse todavía más.

La estrategia más usada se conoce como *selección del pivot por mediana de tres*, y permite eliminar por completo la posibilidad de caer en el peor caso, incluso para distribuciones de datos muy malas o adversas. Esta variante consiste en seleccionar como pivot al *valor mediano* entre *el primero*, *el último* y *el central* del subarreglo analizado (la *mediana* de un conjunto de *n* elementos, es el valor *x* tal que la mitad de los elementos del conjunto son menores a *x*, y la otra mitad son mayores). La forma obvia de obtener el valor mediano de un conjunto es ordenar ese conjunto y luego limitarse a tomar el valor que haya quedado en el centro [1].

En la implementación del *Quicksort por Mediana de Tres*, en cada partición se toma el valor del extremo izquierdo, el valor del extremo derecho y el valor central de la partición, se los ordena entre ellos, y se toma como pivot al que finalmente haya quedado en la casilla central. De esta forma, se garantiza que nunca se tomará como pivot al mayor o al menor, evitando el peor caso antes mencionado. También se suele aconsejar que la *mediana de tres* se tome entre tres elementos seleccionados aleatoriamente dentro de cada partición, lo cual efectivamente lleva a tiempos de ejecución ligeramente mejores.

Por otra parte, se han ensayado otras variantes tomando como pivot al *valor mediano entre cinco* o más componentes del subarreglo, pero las pruebas de rendimiento no han mostrado una mejora significativa en el algoritmo (además de ser más complicadas de programar). Por lo tanto, la estrategia de *selección de pivot por mediana de tres* se ha consolidado como una de las más usadas en la implementación del *Quicksort* (tomando o no los tres elementos en forma aleatoria).

Para finalizar este breve análisis, mostramos la implementación del algoritmo *Quicksort por Mediana de Tres*, tomando la mediana entre el primero, el central y el último de cada partición. Dejamos para los alumnos la implementación con la variante de tomar los tres elementos en forma aleatoria:

Problema 62.) *Implementar el algoritmo Quicksort, pero de forma que la selección del pivot en cada partición se realice por la técnica de la* ***Mediana de Tres****.*

Discusión y solución: Lo único que debemos definir es la forma en que puede obtenerse la mediana entre los tres elementos seleccionados (el primero, el central y el último) de cada partición. El resto del algoritmo es exactamente el mismo que ya hemos mostrado en esta misma Ficha.

Si el arreglo a ordenar es *v*, y la partición en la que se debe tomar el pivot está delimitada por los índices *izq* y *der*, entonces la siguiente función simple reordena los tres elementos *v[izq]*, *v[der]* y *v[central]*, dejando en *v[central]* al valor mediano (que finalmente será retornado):

```
def get_pivot_m3(v, izq, der):
    # calculo del pivot: mediana de tres...
    central = (izq + der) // 2
    if v[der] < v[izq]:
        v[der], v[izq] = v[izq], v[der]
    if v[central] < v[izq]:
        v[central], v[izq] = v[izq], v[central]
    if v[central] > v[der]:
        v[central], v[der] = v[der], v[central]
    return v[central]
```

Los detalles de funcionamiento de esta función son simples y directos, por lo que dejamos su análisis para el estudiante. El algoritmo completo se implementa con las siguientes funciones (incluida la anterior) que forman parte del proyecto *[F31] Divide y Venceras* que acompaña a esta Ficha:

```
def quick_sort_m3(v):
    # ordenamiento Quick Sort
    quick_m3(v, 0, len(v) - 1)


def quick_m3(v, izq, der):
    x = get_pivot_m3(v, izq, der)

    i, j = izq, der
    while i <= j:

        while v[i] < x and i < der:
            i += 1

        while x < v[j] and j > izq:
            j -= 1
```

```
            if i <= j:
                v[i], v[j] = v[j], v[i]
                i += 1
                j -= 1

        if izq < j:
            quick(v, izq, j)

        if i < der:
            quick(v, i, der)


    def get_pivot_m3(v, izq, der):
        # calculo del pivot: mediana de tres...
        central = int((izq + der) / 2)
        if v[der] < v[izq]:
            v[der], v[izq] = v[izq], v[der]
        if v[central] < v[izq]:
            v[central], v[izq] = v[izq], v[central]
        if v[central] > v[der]:
            v[central], v[der] = v[der], v[central]
        return v[central]
```

El programa *main()* del proyecto *[F31] Divide y Venceras* es esencialmente el mismo que ya se mostró en la Ficha de Ordenamiento: un menú que permite crear un arreglo de tamaño *n* con valores aleatorios, y numerosas opciones para ordenar el arreglo con cualquiera de los algoritmos que hemos visto. La diferencia es que ahora se incorpora una opción más para lanzar el *Quicksort por Mediana de Tres*.

Bibliografía

[1] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.

[2] Python Software Foundation, "Python Documentation," 2020. [Online]. Available: https://docs.python.org/3/. [Accessed 24 February 2016].

[3] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

[4] V. Frittelli, R. Teicher, M. Tartabini, J. Fernández and G. F. Bett, "Gestión de Gráficos (en Java) para Asignaturas de Programación Introductiva," in *Libro de Artículos Presentados en I Jornada de Enseñanza de la Ingeniería - JEIN 2011*, Buenos Aires, 2011.

[5] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[6] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[7] Wikipedia, "Ackermann function" 2015. [Online]. Available: https://en.wikipedia.org/wiki/Ackermann_function.

[8] Y. Langsam, M. Augenstein and A. Tenenbaum, Estructura de Datos con C y C++, México: Prentice Hall, 1997.