Título del documento: *Algoritmo QuickSort*

Autor: Wilmer Sneyder Florez Peña

Jose Ronaldo Garzon

Andrés Contreras

Institución: Universidad de Pamplona  
Fecha: 2/12/2024

**Tabla de contenido**

1. Introducción
2. Planteamiento del problema
3. Justificación
4. Objetivos
5. Metodología

**Lista de figuras**

* **Ilustración 1. Interfaz de inicio del algoritmo QuickSort**  
  Descripción de la interfaz principal donde el usuario interactúa con el algoritmo.
* **Ilustración 2. Resultado al presionar "Iniciar aleatorio"**  
  Muestra el estado del sistema después de presionar el botón que genera una lista aleatoria.
* **Ilustración 3. Resultado del algoritmo QuickSort ejecutado**  
  Pantalla mostrando el arreglo ordenado tras ejecutar el algoritmo QuickSort.
* **Ilustración 4. Resultado al presionar "Lista Personalizada"**  
  Muestra el estado del sistema después de que el usuario seleccione una lista personalizada.
* **Ilustración 5. Lista Personalizada Creada**  
  Representa la interfaz con la lista personalizada que el usuario ha creado.
* **Ilustración 6. Pantalla "Mostrar Resultados"**  
  Muestra los resultados obtenidos tras la ejecución del algoritmo, incluyendo el tiempo de ejecución.
* **Ilustración 7. Representación gráfica del número de iteraciones de QuickSort**   
  Gráfico que ilustra el número de iteraciones realizadas por QuickSort al ordenar un arreglo de tamaño 50

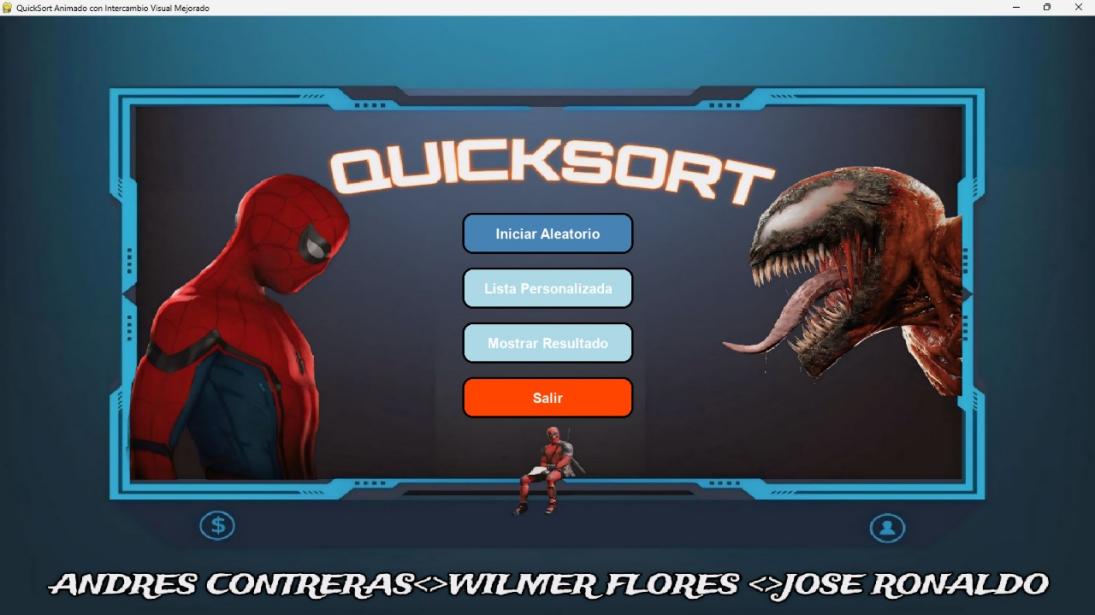
P

Ilustración 1 Interfaz de inicio del algoritmo Quicksort

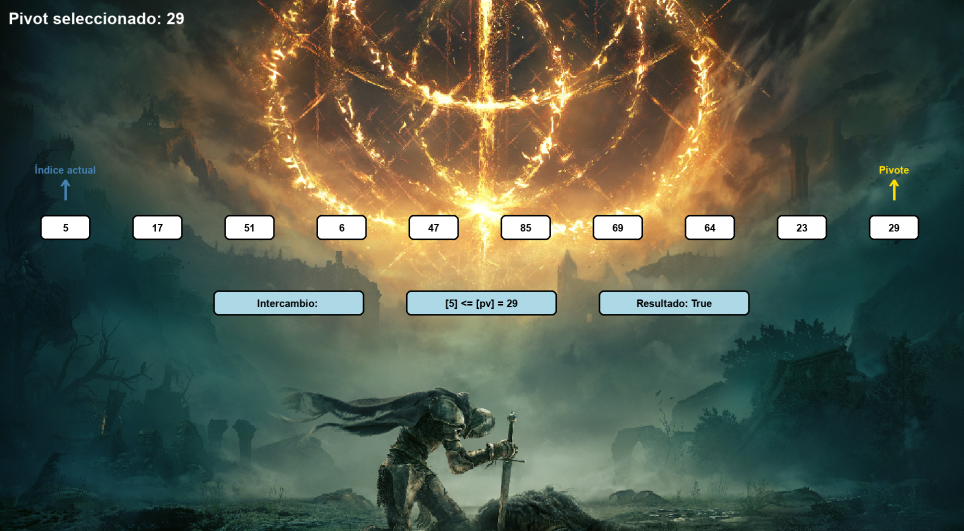


Ilustración 2 Resultado al presionar iniciar aleatorio

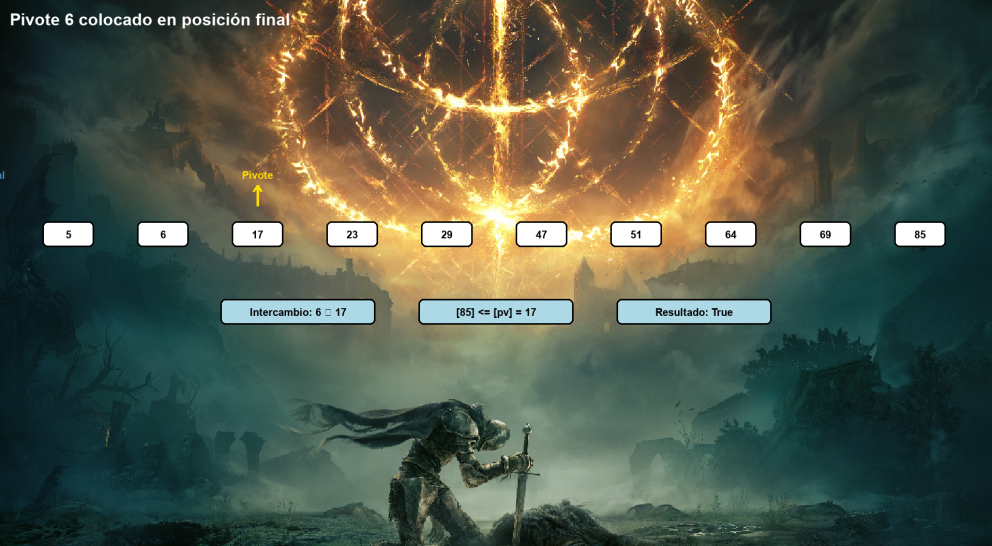


Ilustración 3 Resultado de algoritmo QuickSort ejecutado



Ilustración 4 Resultado al presionar Lista Personalizada

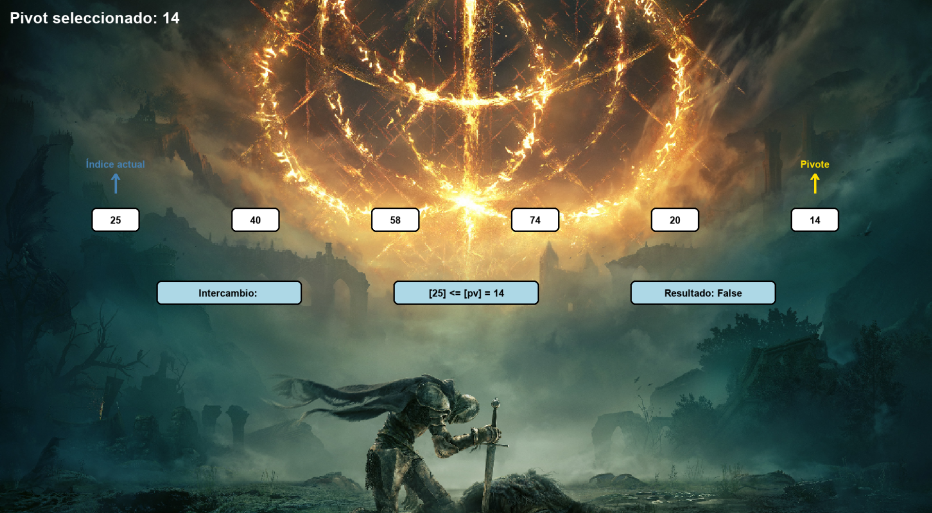


Ilustración 5 Lista Personalizada Creada



Ilustración 6 Pantalla Mostrar Resultados

**Resumen**  
El documento presenta una descripción detallada del algoritmo QuickSort, destacando su eficiencia y aplicación en la ordenación de datos. Este algoritmo es conocido por su capacidad para manejar grandes volúmenes de información en comparación con otros métodos de ordenación, gracias a su enfoque de "divide y vencerás". Se explica su funcionamiento, las ventajas que ofrece en términos de complejidad temporal y espacial, y cómo se implementa de manera práctica en diferentes lenguajes de programación.

**Introducción**  
El algoritmo QuickSort es uno de los más utilizados en la ordenación de grandes conjuntos de datos debido a su eficiencia y rapidez en comparación con otros métodos clásicos como el algoritmo de burbuja o el ordenamiento por inserción. En este trabajo, se aborda su funcionamiento, su aplicabilidad en distintas áreas y su rendimiento en escenarios de grandes bases de datos. El propósito de este estudio es proporcionar una comprensión completa de este algoritmo, analizando su complejidad y las mejores prácticas para su implementación.

**Planteamiento del problema**  
El problema central de este trabajo es el estudio y análisis del algoritmo QuickSort, con el objetivo de determinar su eficiencia frente a otros algoritmos de ordenación más sencillos y explorar los factores que influyen en su rendimiento, como el tamaño de los datos y la elección del pivote.

**Justificación**  
El algoritmo QuickSort es fundamental en áreas como la programación de bases de datos, la computación de grandes volúmenes de datos y la inteligencia artificial. Dado que la eficiencia en la manipulación de datos es crucial, estudiar su comportamiento y optimización es de gran relevancia para mejorar el rendimiento de sistemas informáticos en aplicaciones donde la rapidez en la ordenación es crítica.

**Objetivos**

* Objetivo general: Analizar el algoritmo QuickSort y evaluar su eficiencia en comparación con otros algoritmos de ordenación.
* Objetivos específicos:
  1. Describir el funcionamiento del algoritmo QuickSort.
  2. Realizar una comparación entre QuickSort y otros algoritmos de ordenación.
  3. Evaluar el rendimiento del algoritmo QuickSort en distintas situaciones de ordenación.

**Herramientas utilizadas**:

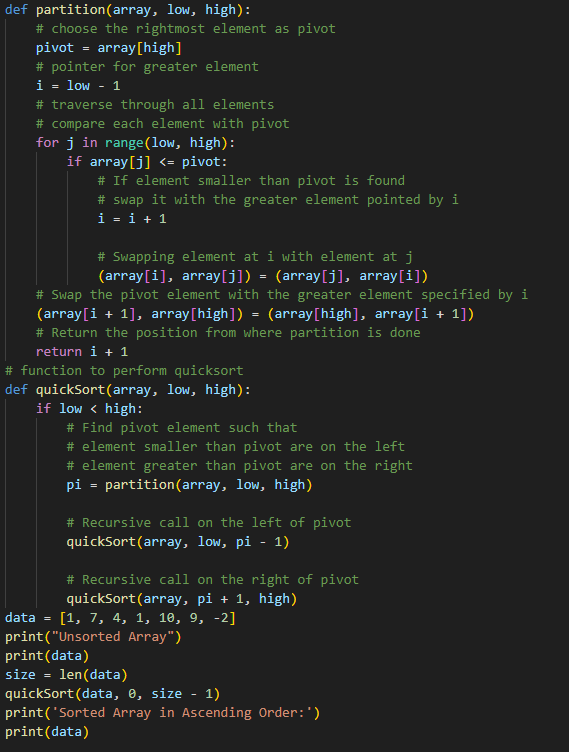
* **Python:** Lenguaje de programación utilizado para la implementación del algoritmo.
* **Librerías de Python**: Se usaron las librerías para la visualización de gráficos como pygame, y randmom para la creación de listas aleatorias.
* **Entorno de desarrollo**: Se trabajó en un entorno de desarrollo como visual studio Code.

**Metodología**  
La metodología adoptada para este trabajo tiene como objetivo analizar y evaluar el algoritmo QuickSort desde una perspectiva teórica y práctica, considerando su eficiencia, complejidad y desempeño en distintos escenarios. A continuación, se describen las fases y técnicas empleadas:

1. **Estudio teórico del algoritmo QuickSort**  
   Se realizó una revisión bibliográfica para comprender los fundamentos teóricos del algoritmo QuickSort, incluyendo:
   * Su enfoque basado en la técnica de *divide y vencerás*.
   * Las principales variantes del algoritmo, como la selección del pivote en diferentes posiciones (primer elemento, último elemento, pivote aleatorio o pivote mediano).
   * Su complejidad computacional en el mejor caso, caso promedio y peor caso.

Las fuentes consultadas incluyeron, artículos académicos y documentación en línea especializada.

1. **Implementación del algoritmo QuickSort**  
   Se implementó el algoritmo QuickSort en Python siguiendo un enfoque recursivo estándar. El pseudocódigo utilizado como base para la implementación es el siguiente:



*  Selección **del pivote**: Se probó con diferentes estrategias para la selección del pivote (primer elemento, último elemento, pivote aleatorio, y pivote mediano) y se analizaron sus impactos en el rendimiento.
* **Optimización de la partición**: Se utilizó una implementación eficiente para dividir el arreglo en subarreglos menores y mayores al pivote.

3.**Generación de conjuntos de datos de prueba**  
Para evaluar el rendimiento del algoritmo, se generaron diferentes tipos de conjuntos de datos:

* **Datos aleatorios**: Conjunto de números generados aleatoriamente.
* **Datos ordenados ascendentemente**: Para observar el peor caso en algunos escenarios.
* **Datos ordenados descendentemente**: Para identificar comportamientos en casos no óptimos.
* **Datos parcialmente ordenados**: Mezcla de elementos ordenados y desordenados.

Los tamaños de los conjuntos de datos variaron entre 10⁴, 10⁵ y 10⁶ elementos para evaluar el desempeño del algoritmo en condiciones de pequeña, mediana y gran escala.

4.**Pruebas de rendimiento**  
Cada conjunto de datos fue ordenado utilizando la implementación de QuickSort. Se midió el tiempo de ejecución en cada caso utilizando la librería time de Python. Los tiempos de ejecución fueron promediados después de realizar múltiples ejecuciones para obtener resultados más confiables.

**5.Visualización de resultados**  
Los resultados obtenidos se analizaron utilizando gráficos generados con la librería matplotlib. Los gráficos mostraron:

* El tiempo de ejecución en función del tamaño del conjunto de datos.
* Comparaciones entre las diferentes estrategias de selección del pivote en QuickSort.

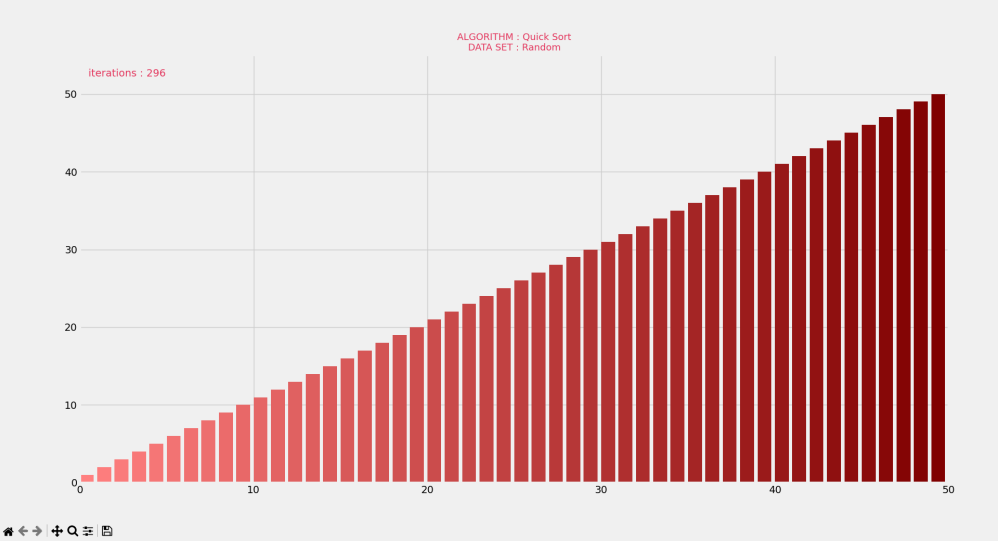


Ilustración 7 Representación Gráfica número de iteraciones de QuickSort

**6Análisis cualitativo y cuantitativo**  
Finalmente, se realizó un análisis detallado que incluyó:

* Identificación de las condiciones que maximizan la eficiencia de QuickSort.
* Evaluación de su comportamiento en escenarios adversos (como datos ya ordenados).
* Comparación de la complejidad teórica con los resultados experimentales para verificar la correspondencia.

**Conclusiones**

El análisis realizado sobre el algoritmo QuickSort permitió obtener las siguientes conclusiones:

1.**Eficiencia y aplicabilidad**:  
QuickSort destaca por su eficiencia en la ordenación de datos gracias a su enfoque basado en *divide y vencerás*. Es especialmente adecuado para grandes volúmenes de datos en entornos donde la velocidad es crucial. Su complejidad promedio de O(nlogn) lo convierte en una opción preferida frente a algoritmos más simples como el Bubble Sort (Cormen et al., 2009).

2.**Importancia de la selección del pivote**:  
La selección del pivote es un factor crítico para garantizar el rendimiento óptimo del algoritmo. Estrategias como la elección aleatoria o el pivote mediano tienden a evitar el peor caso O(n^2) que ocurre con arreglos ordenados cuando se selecciona el primer o último elemento como pivote (Hoare, 1962).

3. **Comparación con otros algoritmos**:  
Aunque QuickSort es más rápido en la mayoría de los casos, otros algoritmos como Merge Sort son más consistentes en términos de tiempo de ejecución debido a su complejidad garantizada de O(nlogn). Sin embargo, Merge Sort requiere mayor uso de memoria, mientras que QuickSort opera *in-place*, siendo más eficiente en términos de espacio (Sedgewick & Wayne, 2011).

4. **Limitaciones y mejoras**:  
QuickSort puede experimentar problemas de eficiencia en escenarios específicos, como arreglos ya ordenados o parcialmente ordenados, si no se implementan estrategias adecuadas para la selección del pivote. Implementaciones híbridas, como Introsort, combinan QuickSort con otros algoritmos para superar estas limitaciones (Heineman et al., 2008).

Link de la presentacion:https://www.canva.com/design/DAGYvmwQdzE/9ba-06FRS1VMEneRqfbozQ/edit?utm\_content=DAGYvmwQdzE&utm\_campaign=designshare&utm\_medium=link2&utm\_source=sharebutton

**Referencias**

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press
2. Documentation Python. (n.d.). Sorting Algorithms. [https://docs.python.org](https://docs.python.org" \t "_new).
3. Hoare, C. A. R. (1962). Quicksort. The Computer Journal, 5(1), 10–16.
4. Heineman, G. T., Pollice, G., & Selkow, S. (2008). Algorithms in a Nutshell: A Practical Guide. O'Reilly Media.
5. Skiena, S. S. (2012). The Algorithm Design Manual. Springer.
6. Sedgewick, R., & Wayne, K. (2011). Algorithms. Addison-Wesley Professional.