Universidad del Caribe

2000 CANCUN, QUINTANA ROO, MÉXICO

CONOCIMIENTO Y CULTURA PARA EL DESARROLLO HUMANO

Reporte:

# Ejercicio Final

Asignatura:

**tecnicas algoritmicas**


**Torres Bolaños Uzli Enrique**

MATRÍCULA: **200300655**

PROGRAMA EDUCATIVO:ingeniería en datos e inteligencia organizacional


Presentado a:

**PROF. Emmanuel Morales Saavedra**

Cancún, Quintana Roo

Abril XX, 2023

## Introducción:

En este proyecto, me propuse un desafío interesante: implementar un solucionador de Sudoku que utilizará diferentes técnicas de ordenamiento para hacer más eficiente el proceso de llenado del tablero. Decidí trabajar con algoritmos de ordenamiento populares como QuickSort, Insertion Sort y Merge Sort. La idea era simple: en cada celda vacía del Sudoku, identificar los números candidatos (aquellos que podrían ir en esa celda sin violar las reglas) y ordenarlos antes de probarlos uno por uno. Esto, en teoría, podría hacer que el proceso de resolución fuera un poco más organizado y tal vez incluso más rápido.

Además de las versiones con los algoritmos de ordenamiento, también incluí una opción básica que no ordena los candidatos en absoluto. ¿Por qué? Porque quería comparar y entender cómo influye realmente el ordenamiento en la eficiencia del proceso. Muchas veces, pensamos que agregar pasos como ordenar puede mejorar las cosas, pero no siempre es así, y este proyecto era una oportunidad perfecta para experimentar con esta idea.

## Justificación de las Técnicas Seleccionadas:

Cuando me enfrenté al reto de implementar un solucionador de Sudoku, me di cuenta de que elegir los algoritmos de ordenamiento correctos era crucial para entender cómo podrían afectar el rendimiento del proceso. Por eso, seleccioné un conjunto de técnicas conocidas, cada una con sus propias ventajas y características, que me parecieron interesantes de probar en este contexto. Aquí les cuento por qué elegí cada una:

QuickSort: Este algoritmo es conocido por ser rápido y eficiente en la mayoría de los casos, con una complejidad promedio de $O(n \log n)$
O(nlogn). Aunque su rendimiento depende de cómo se elija el pivote, suele comportarse muy bien en conjuntos pequeños como los candidatos de Sudoku. Además, es uno de esos algoritmos clásicos que siempre quise probar en algo práctico.

Insertion Sort: Aunque este algoritmo tiene una complejidad de $O(n2)$O(n 2 ), es bastante eficiente para listas pequeñas o cuando los elementos ya están casi ordenados. Me pareció interesante incluirlo porque, aunque no sea el más rápido en teoría, puede sorprender en escenarios específicos, como los pequeños grupos de candidatos que se generan en cada celda del Sudoku.

Merge Sort: Este es el caballo de batalla de los algoritmos de ordenamiento. Con su complejidad garantizada de $O(n\log n)$ O(nlogn) en todos los casos, quería ver si su

estabilidad y consistencia harían alguna diferencia significativa en el contexto de un Sudoku. Además, su estructura recursiva encaja bien con la lógica que ya estaba usando para resolver el tablero.

Sin Ordenar: Para completar el experimento, incluí una versión que simplemente no ordena los candidatos antes de probarlos. Esto sirve como punto de referencia para evaluar si el esfuerzo adicional de ordenar realmente tiene un impacto en el tiempo total de resolución. Es como la línea base del experimento: ¿realmente necesitamos ordenar o el Sudoku puede resolverse igual de bien sin este paso?

Lo que quería lograr con estas técnicas era explorar cómo influye la organización de los candidatos en el rendimiento general del solucionador. Aunque podría parecer que ordenar siempre es mejor, en realidad, este proyecto era una oportunidad perfecta para cuestionar esa idea y descubrir si vale la pena complicar el proceso o si la simplicidad termina ganando.

## Análisis de la Complejidad Computacional:

*Resolución del Sudoku*

El proceso consiste en explorar cada celda vacía, probar cada candidato disponible y avanzar a la siguiente celda si el candidato es válido. Si un intento falla más adelante, el algoritmo retrocede y prueba el siguiente candidato. Este enfoque funciona bien en tableros con restricciones definidas, pero el tiempo requerido aumenta rápidamente cuando hay menos restricciones.

Ordenamiento de Candidatos
El siguiente paso importante es organizar los posibles candidatos de cada celda antes de probarlos. Aunque en un Sudoku estándar el número máximo de candidatos por celda es nueve, el método de ordenamiento que se elija puede influir en el rendimiento general, especialmente si consideramos que esta operación se realiza repetidamente durante el proceso de resolución.

QuickSort:
Este algoritmo es muy popular debido a su eficiencia promedio. Suele ser rápido y funciona bien para listas pequeñas como las de los candidatos del Sudoku. Sin embargo, su rendimiento puede disminuir en ciertos casos, dependiendo de cómo se seleccione el elemento pivote para dividir la lista. Aunque esto podría ser un problema con listas más grandes, en este proyecto, el tamaño limitado de las listas reduce significativamente el impacto negativo.

```cpp
void quickSort(vector<int>& nums, int low, int high) {
    if (low < high) {
        int pivot = nums[high];
        int i = low - 1;
        for (int j = low; j <= high - 1; j++) {
            if (nums[j] <= pivot) {
                i++;
                swap(nums[i], nums[j]);
            }
        }
        swap(nums[i + 1], nums[high]);
        int pi = i + 1;

        quickSort(nums, low, pi - 1);
        quickSort(nums, pi + 1, high);
    }
}
```

Insertion Sort:
Es un método sencillo que ordena los elementos comparándolos y ubicándolos en su lugar correcto uno por uno. Aunque no es el más rápido en teoría, puede ser sorprendentemente eficiente para listas pequeñas o que ya están casi ordenadas. En este contexto, donde las listas de candidatos no suelen ser muy largas, me pareció interesante probarlo como alternativa.

```cpp
void insertionSort(vector<int>& nums) {
    for (size_t i = 1; i < nums.size(); i++) {
        int key = nums[i];
        int j = i - 1;
        while (j >= 0 && n
            nums[j + 1] = nums[j];
            j--;
        }
        nums[j + 1] = key;
    }
}
```

std::vector<int> &nums

Merge Sort:

Este es un algoritmo muy confiable que divide las listas en partes más pequeñas, las ordena y luego las combina. Es estable y garantiza un rendimiento consistente en todos los casos. Sin embargo, dado que las listas de candidatos son pequeñas, su ventaja frente a otros métodos puede no ser tan significativa en este escenario específico.

```cpp
void merge(vector<int>& nums, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++) L[i] = nums[left + i];
    for (int i = 0; i < n2; i++) R[i] = nums[mid + 1 + i];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) nums[k++] = L[i++];
        else nums[k++] = R[j++];
    }

    while (i < n1) nums[k++] = L[i++];
    while (j < n2) nums[k++] = R[j++];
}

void mergeSort(vector<int>& nums, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(nums, left, mid);
        mergeSort(nums, mid + 1, right);
        merge(nums, left, mid, right);
    }
}
```

Sin Ordenar:

Por último, decidí incluir una variante del solucionador que no aplica ningún método de ordenamiento. Esta versión funciona como un punto de referencia para medir si ordenar realmente influye en el tiempo total de resolución del tablero. Si el tiempo de ejecución no cambia mucho, puede que el esfuerzo adicional de ordenar no sea necesario.

Impacto General

Aunque el impacto del ordenamiento de los candidatos es mínimo desde un punto de vista teórico, el efecto puede notarse en términos prácticos, especialmente al medir el tiempo real de ejecución. En un Sudoku estándar, donde las listas de candidatos son pequeñas, los beneficios de ordenar pueden no ser evidentes inmediatamente. Sin embargo, como este paso se repite para cada celda vacía,

cualquier mejora en el rendimiento del ordenamiento puede acumularse y tener un efecto positivo en el tiempo total.

Al final, este análisis no sólo me ayudó a comprender las diferencias entre los métodos de ordenamiento, sino que también sirvió para cuestionar si realmente vale la pena complicar el proceso con un paso adicional o si el solucionador podría funcionar igual de bien sin ordenar. Este balance entre teoría y práctica fue uno de los aspectos más interesantes del proyecto.