Algoritmos y Estructuras de Datos

Cátedra Juárez

INFORME DE COMPLEJIDAD ALGORÍTMICA

Nombre del grupo: AFIP

Integrantes:

- NARVÁEZ, GABRIEL - 111432
- SILVA, GUILLERMO - 109777
- FARIÑA, ALEX - 112438
- RIVERO, CANDELA- 110339

FIUBA 2024

Resumen

Se tuvo como objetivo comparar teóricamente y experimentalmente la complejidad logarítmica del “treesort” (método de ordenamiento). Se pregunto ¿Son iguales los resultados teóricos y experimentales?, en caso de no ser iguales ¿tienen alguna similitud?. La metodología cuantitativa fue usada en gran medida para obtener datos, estos datos fueron favorables para concluir que el treesort teórico y experimental son similares.

Introducción

El objetivo principal es comparar entre lo empírico y teórico de la complejidad logarítmica del método de ordenamiento “treesort”. Se mantiene la suposición que el método de ordenamiento “treesort” deberá dar resultados similares entre lo empírico e hipotético, es decir, que de alguna manera se van a poder comparar entre ambos.

La complejidad logarítmica se refiere al tiempo de ejecución que necesita un algoritmo para resolver algún problema y en base a esto determinar si es eficiente. El algoritmo es una serie de pasos para solucionar un problema.

El “treesort” es un método de ordenamiento que se basa en un árbol binario de búsqueda (ABB), la ejecución de este es agregar los valores del “arreglo” o “vector” y después ordenarlos. La complejidad del treesort:

- Mejor de los casos: O(nlog(n))
- Caso promedio: O(nlog(n))
- Peor de los casos: $O(n^2)$

El árbol binario de búsqueda (ABB) es un tipo de estructura de datos que tiene forma de un árbol, donde coexisten raíz(padre) e hijos (2 por binario), esto es eficiente cuando se quiera encontrar datos debido a como esta armado, ya que va dividiendo el problema en 2 hasta llegar al resultado.

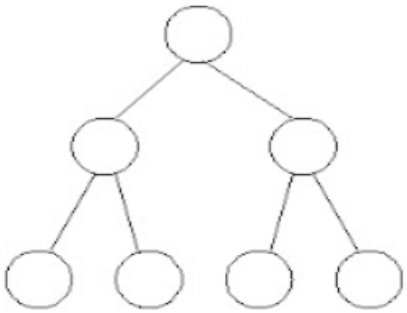

Imagen 1 : Estructura de un ABB

https://i.sstatic.net/e2Chw.png

Desarrollo

Para empezar, de un “arreglo” o “vector” con números aleatorios desordenados, se agregaran a un ABB, para después incorporarlos de manera ordenada en otro “vector”. Se usaron “clases” que se creó para mantener el conteo de las operaciones que se realizo a lo largo de cada medición, las clases son:

- Vector: Encargado del alta de datos ya ordenados, que se obtiene al ordenar el ABB
- ABB: Obtienen los datos del “vector” o “arreglo” que se desea ordenar, para armar el arbol.
- Nodo: Conforma a la clase ABB aquí se almacenan los datos con la información.

Para obtener un “arreglo” aleatorio, se agrega elementos del 1 hasta n (n múltiplos de 10, desde 10 hasta 1000) y después se mezclan. Ejemplo:

1	2	3	4	5

4	1	5	2	3

Después se inicia con el conteo del tiempo que tarda el algoritmo del treesort, termina cuando todos los números del arreglo están ordenados. Se guarda en un archivo “.csv” la cantidad de datos, tiempo de ejecución del algoritmo y la cantidad de operaciones que hizo el algoritmo.

```cpp
#include <iostream>
#include <algorithm>
#include <random>
#include <chrono>
#include "Vector.hpp"
#include "Diccionario.hpp"
#include <fstream>


void asignar_datos_vector(Vector<int>& vector, int cantidad){
    for(int i = 1; i <= cantidad; i++){
        vector.alta(i);
    }
    std::random_device rd;
    std::mt19937 g(rd());

    std::shuffle(&vector[0], &vector[vector.tamanio() - 1] + 1, g);
}

void treesort(Vector<int>& ordenado, Diccionario<int>& arbol, Vector<int> vector, int& contador){
    contador++;
    for(size_t i = 0; i < vector.tamanio(); i++){

        arbol.alta(vector[i]);
    }
    contador++;
    ordenado = arbol.inorder();
}

int main() {
    Vector<std::chrono::milliseconds> tiempo;
    std::fstream file;
    file.open("archivo.csv", std::fstream::out);
    if(file.is_open()){
        for(int i = 1; i <= 100; i++){
            Vector<int> numeros;
            Vector<int> respuesta;
            Diccionario<int> tree;
            int contador;

            asignar_datos_vector(numeros, 10*i);
            auto inicio = std::chrono::high_resolution_clock::now();

            treesort(respuesta, tree, numeros, contador);

            auto final = std::chrono::high_resolution_clock::now();
            std::chrono::duration<double> duracion = final - inicio;
            contador = tree.obtener_cantidad_operaciones();
            file << 10*i;
            file << ",";
            file << duracion.count();
            file << ",";
            file << contador;
            file << std::endl;
        }
    }
}
```

Imagen 1: main.cpp

Por ultimo se ejecuta 1000 veces el programa anterior, y en base a eso se obtiene un promedio de la complejidad del treesort (mejor caso, peor caos y caso promedio). Ademas se usa la biblioteca “matplotlib” para mostrar 2 gráficos del tiempo en función de la cantidad de datos que tiene el “arreglo”, el primero con los tiempos desordenados y el segundo en orden.

```python
import matplotlib.pyplot as plt
import csv
import subprocess

promedio_diferencia = []
promedio_mejor = []
promedio_peor = []
promedio_promedio = []

for i in range (1000):
    subprocess.run('/home/gabriel/Desktop/ComplejidadAlgoritmica/ejecutable')
    with open('archivo.csv', newline='') as archivo:
        spamreader = csv.reader(archivo, delimiter=',', quotechar='|')
        vecY_tiempo = []
        vecY_conteo = []
        vecX = []
        promedio = 0
        peor_caso = 0
        mejor_caso = 100
        for linea in spamreader:
            vecX.append(int(linea[0]))
            vecY_tiempo.append(float(linea[1]))
            vecY_conteo.append(float(linea[2]))
        for j in range(100):
            promedio = promedio + vecY_tiempo[j]
            if(peor_caso < vecY_tiempo[j]):
                peor_caso = vecY_tiempo[j]
            if(mejor_caso > vecY_tiempo[j]):
                mejor_caso = vecY_tiempo[j]
        promedio = promedio/1000
        promedio_peor.append(peor_caso)
        promedio_promedio.append(promedio)
        promedio_mejor.append(mejor_caso)
        if(promedio > mejor_caso):
            promedio_diferencia.append(promedio - mejor_caso)
        else:
            promedio_diferencia.append(mejor_caso - promedio)
        if(i == 999):
            plt.plot(vecX, vecY_tiempo)
            plt.title("Complejidad Temporal")
            plt.xlabel("Cantidad de datos (números)")
            plt.ylabel("Tiempo (milisegundos)")
            plt.grid(True)
            plt.show()
            vecY_tiempo.sort()
            plt.plot(vecX, vecY_tiempo)
            plt.title("Complejidad Temporal")
            plt.xlabel("Cantidad de datos (números)")
            plt.ylabel("Tiempo (milisegundos)")
            plt.grid(True)
            plt.show()

mejor_caso = 0
peor_caso = 0
promedio = 0
acotacion = 0
for i in range(1000):
    mejor_caso = mejor_caso + promedio_mejor[i]
    peor_caso = peor_caso + promedio_peor[i]
    promedio = promedio + promedio_promedio[i]
    acotacion = acotacion + promedio_diferencia[i]

mejor_caso = mejor_caso/1000
peor_caso = peor_caso/1000
promedio = promedio/1000
acotacion = acotacion/1000

print(mejor_caso)
print(peor_caso)
print(promedio)
print(acotacion)
```

Imagen 2: graficos.py

Resultados

Lo que se obtuvo al experimentar la complejidad temporal del “treesort” fue que en peor de los casos va a tardar un tiempo de 0.00022339805800000015 , en el mejor de los casos 3.418992999999998e-06 y en promedio 1.0040367472000012e-05. Estos datos observados son un promedio de las 1000 muestras que se ejecutó el programa.

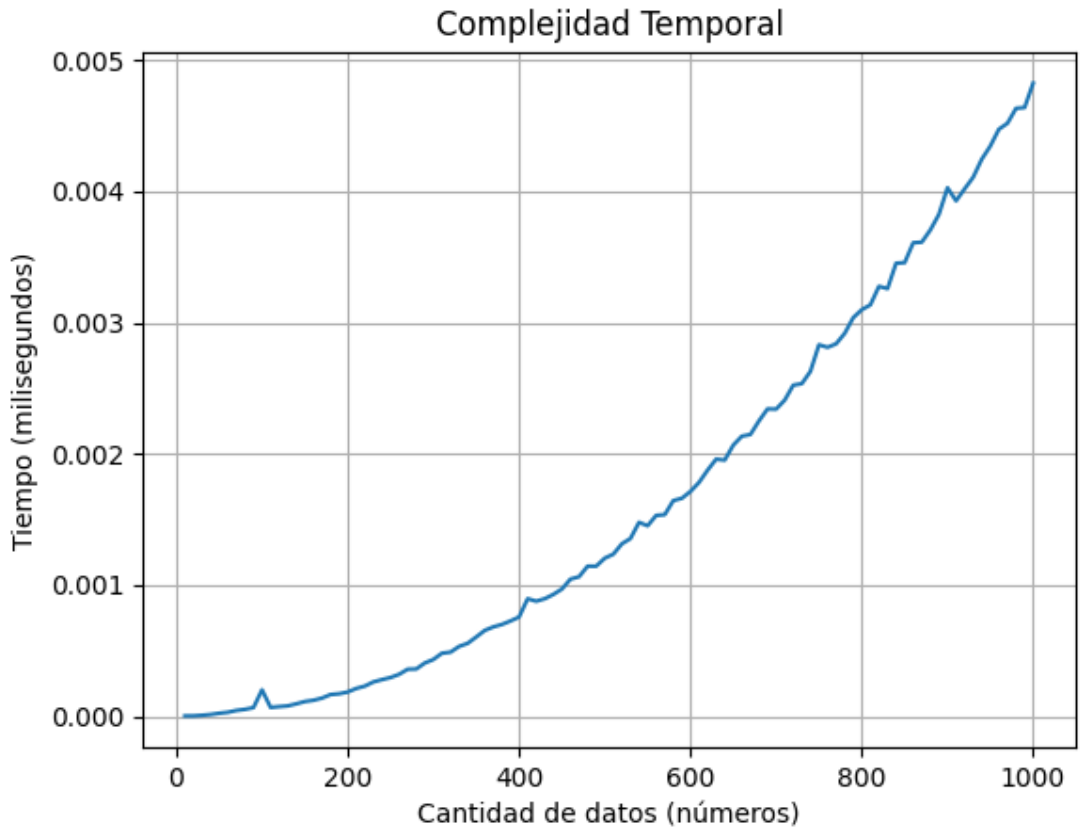

Gráfico 1: Grafico de tiempo de ejecución del treesort en función de la cantidad de datos de un arreglo ordenado. Árbol ABB mal generado.

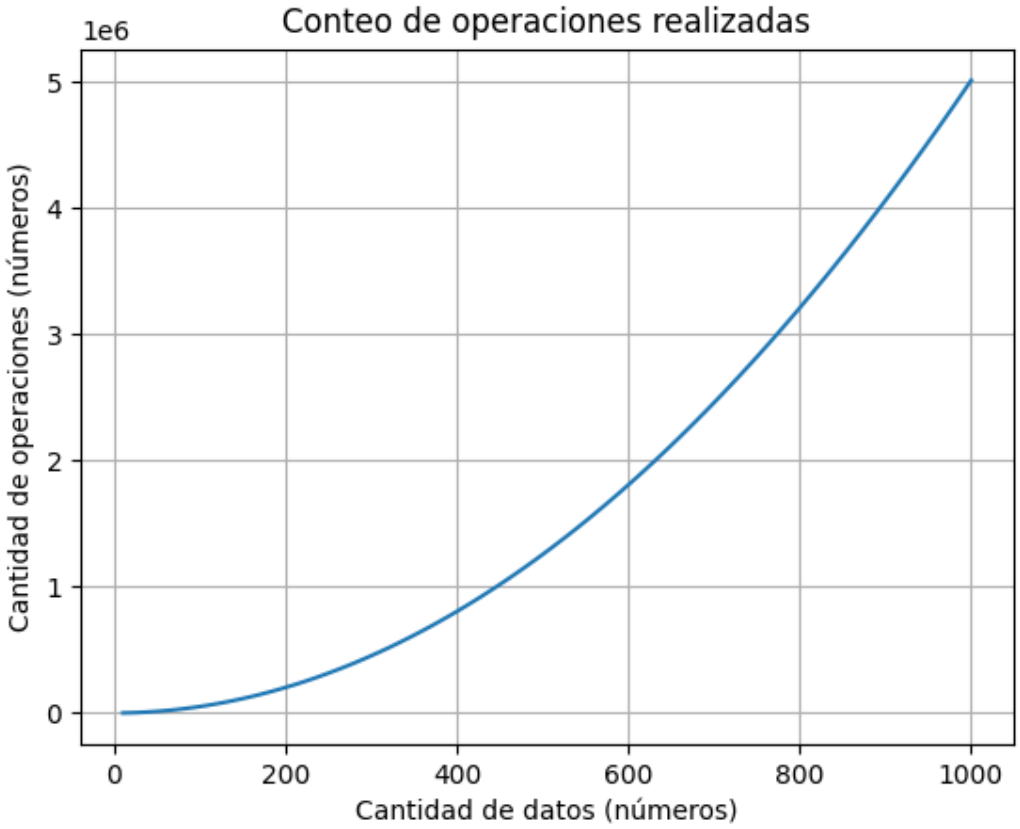

Gráfico 3: Grafico de cantidad de operaciones del treesort en función de la cantidad de datos de un arreglo ordenado. Árbol ABB mal generado.

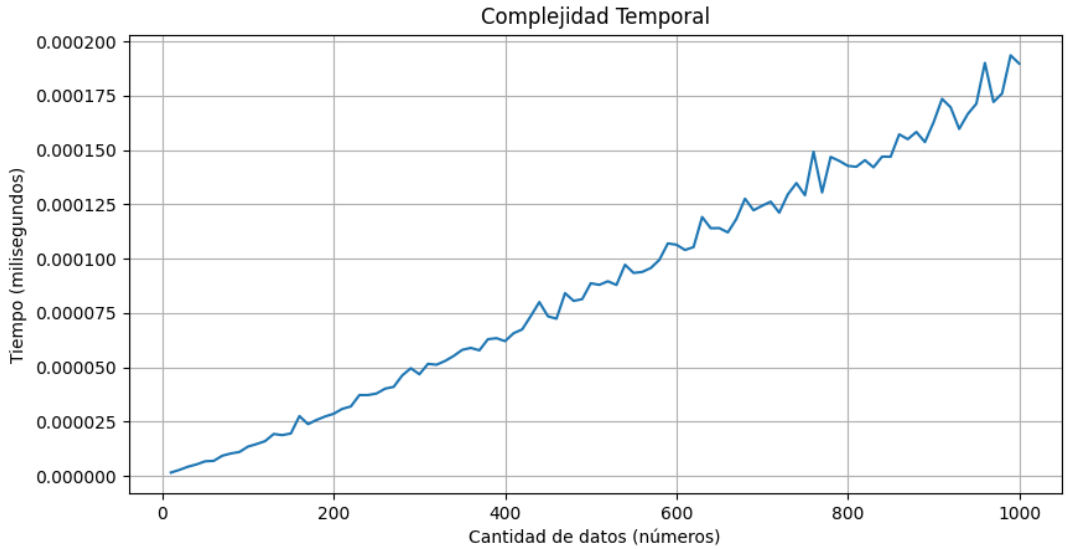

Gráfico 3: Grafico de tiempo de ejecución del treesort en función de la cantidad de datos de un arreglo genérico

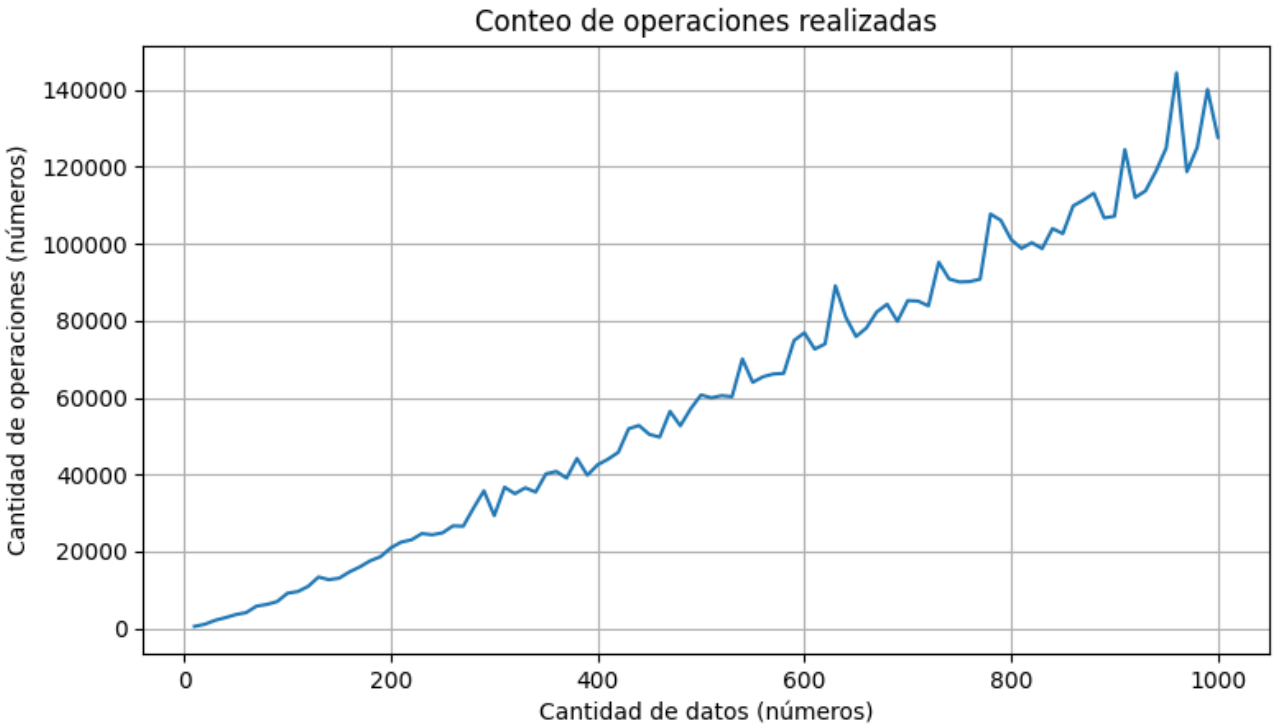

Gráfico 4: Grafico de cantidad de operaciones del treesort en función de la cantidad de datos de un arreglo generico

Análisis de los resultados

La comparación de lo teórico y empírico fueron resultados positivos, según la hipótesis planteada:

	Teórico	Empírico
Mejor caso	nlog(n)	3.418992999999998e-06
Peor caso	n^2	0.00022339805800000015
Caso promedio	nlog(n)	1.0040367472000012e-05

El mejor caso y el caso promedio teóricamente son iguales, pero en la parte experimental es diferente con 6.621374471999997e-06. Se puede asumir que no son diferentes, si no similares debido a que es un numero muy pequeño, en contraste al peor caso que es mayor ambos.

Conclusiones

A grandes rasgos los resultados experimentales en el análisis, son consistentes con las expectativas teóricas. Como se menciona, van a ser similares respecto a lo visto en clases, pero no iguales. Los resultados en el mejor y en caso promedio teóricamente son iguales, sin embargo a partir de los datos experimentales estos difieren ligeramente, debido a eficiencia en la implementación y/o deficiencias del entorno en el cual estamos trabajando. Estas diferencias no quieren decir que nuestros resultados sean erróneos ya que refuerzan la idea.

En cuanto al peor caso lo que podemos intuir es que se podría deber a que el árbol este muy desbalagado tanto, al punto de ser una lista enlazada y en nuestras pruebas experimentales dado que obteníamos las claves de los nodos de manera aleatoria esto producía que las claves no permitan que el árbol se degenere.

Anexo

Departamento de Informática & Universidad de Valladolid. (n.d.). *COMPLEJIDAD ALGORÍTMICA*. https://www.infor.uva.es/~jvalvarez/docencia/tema5.pdf

GeeksforGeeks. (2023, September 11). *Tree Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/tree-sort/

Hernández, H., & Silva, S. (2013). *METODOLOGÍA DE LA INVESTIGACIÓN. In METODOLOGÍA DE LA INVESTIGACIÓN* (p. v). https://www.gob.mx/cms/uploads/attachment/file/133491/METODOLOGIA_DE_INVESTIGACION.pdf