Tarea individual: Act-Integradora-1 Conceptos básicos y algoritmos fundamentales

Arturo Sánchez Rodríguez | A01275427

Grant Keegan | A01700753

28 de marzo del 2023

Programación de estructuras de datos y algoritmos fundamentales (Gpo 850)

Profesor: Dr. Eduardo Arturo Rodríguez Tello

Introducción

En esta actividad, nuestro objetivo es escribir un programa en C++ que tome una lista de 16,000 entradas en un archivo .txt., por medio de algoritmos de ordenamiento (Bubble e insertion sort) ordena esas entradas por fecha, e imprima los resultados en otro archivo .txt.

Pasos a seguir:

1. Investigar sobre los algoritmos de ordenamiento y búsqueda más eficientes para la situación planteada.

2. Diseñar la estructura de datos (clase) para almacenar los datos del archivo.

3. Codificar los algoritmos de ordenamiento y búsqueda.

4. Realizar pruebas con los casos de prueba proporcionados.

5. Desplegar en pantalla los resultados de la comparación entre los algoritmos de ordenamiento.

6. Solicitar al usuario las fechas de inicio y fin del rango de búsqueda.

7. Utilizar el mejor algoritmo de búsqueda para encontrar la información solicitada.

8. Desplegar en pantalla los registros correspondientes al rango de fechas.

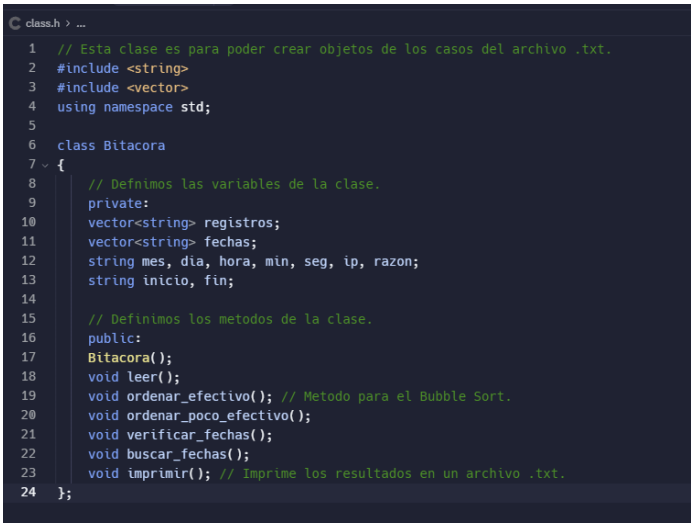
9. Almacenar en un archivo el resultado del ordenamiento de la bitácora completa.

10. Realizar una investigación y reflexión en forma individual de la importancia y eficiencia del uso de los diferentes algoritmos de ordenamiento y búsqueda, contrastando los resultados con el análisis de la complejidad temporal de dichos algoritmos.

11. Generar un documento con toda la información recopilada.

Parte 1: Diseñando la clase

En esta actividad, lo primero que decidimos hacer fue diseñar nuestra clase con la que crearemos las instancias de objetos sobre las entradas del documento bitácora .txt. Creamos la clase ‘Bitácora’ en la que nuestros atributos fueron los 7 datos de los casos de errores (mes, día, hora, minuto, segundo, la dirección IP y la razón del problema).



El objetivo ahora de la función es tomar las más de 16,000 líneas de texto, y usando la clase crear instancias para poder agregar a una lista, con los atributos respectivos. A partir de aquí podremos ordenar las entradas.

La clase “Bitacora” fue diseñada para poder leer, ordenar y buscar información de un archivo de texto. Esta clase contiene los atributos "registros", "fechas", "mes", "dia", "hora", "min", "seg", "ip", "razon", "inicio" y "fin". Estos atributos se utilizan para almacenar los datos leídos del archivo de texto. Además, la clase Bitacora contiene los métodos "leer()", "ordenar_efectivo()", "ordenar_poco_efectivo()", "verificar_fechas()", "buscar_fechas()" e "imprimir()". Estos métodos se usan para leer los datos del archivo de texto, ordenarlos con los algoritmos de ordenamiento Insertion Sort y Bubble Sort, verificar la existencia de fechas específicas, buscar información entre dos fechas específicas y finalmente imprimir los resultados en un archivo de texto.

Parte 2: Escribiendo los algoritmos.

La actividad específica que debemos de crear 2 diferentes tipos de algoritmos para resolver el mismo problema. Por lo que decidimos elegir los algoritmos de insertion y Bubble Sort para trabajar con los objetos creados por la clase en base al archivo.txt.

```
// Este algortimo es para el ordenamiento de Insertion Sort.
void Bitacora::ordenar_poco_efectivo(){
    int comparaciones = 0; // Para contear las comparaciones e imprimirlos.
    int swaps = 0; // Para contear los swaps e imprimirlos.
    for(int i = 0; i < registros.size()-1; i++){
        for(int j = 0; j < registros.size()-i-1; j++){
            comparaciones++;
            if(fechas[j] > fechas[j+1]){
                swaps++;
                string temp1 = fechas[j];
                string temp2 = registros[j];
                fechas[j] = fechas[j+1];
                registros[j] = registros[j+1];
                fechas[j+1] = temp1;
                registros[j+1] = temp2;
            }
        }
    }
    cout << "Algoritmo poco efectivo." << endl;
    cout << "Comparaciones realizadas: " << comparaciones << endl;
    cout << "Swaps realizados: " << swaps << endl;
};
```

El método de ordenamiento Insertion Sort funciona comparando el elemento actual con los anteriores para encontrar el lugar correcto para ella, esto se repite para cada elemento en la lista. El Insertion Sort es un algoritmo de ordenamiento sencillo. No es tan eficiente como otros métodos como Heap Sort, Quick Sort o Merge Sort, pero es muy fácil de comprender y aplicar. Funciona recibiendo una lista de objetos comparables y devolviéndolos ordenados. Esto se realiza a través de un ciclo que comienza desde la segunda posición y termina en la última. En cada iteración, el elemento arreglo[i] es insertado en su lugar correspondiente de la secuencia previamente ordenada arreglo[1...i-1], de modo que se ordena el arreglo.

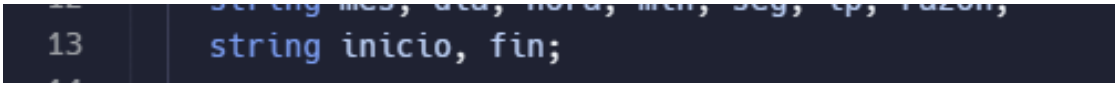
```
// Este algoritmo es para el ordenamiento de Bubble Sort.
void Bitacora::ordenar_efectivo(){
    int comparaciones = 0; // Para contear las comparaciones e imprimirlos.
    int swaps = 0; // Para contear los swaps e imprimirlos.
    for(int i = 1; i < registros.size(); i++){
        for(int j = i; j > 0; j--){
            comparaciones++;
            if(fechas[j] < fechas[j-1]){
                swaps++;
                string temp1 = fechas[j];
                string temp2 = registros[j];
                fechas[j] = fechas[j-1];
                registros[j] = registros[j-1];
                fechas[j-1] = temp1;
                registros[j-1] = temp2;
            }
            else
                break;
        }
    }
    cout << "Algoritmo efectivo." << endl;
    cout << "Comparaciones realizadas: " << comparaciones << endl;
    cout << "Swaps realizados: " << swaps << endl;
};
```

El método de ordenamiento Bubble Sort funciona comparando el elemento actual con el siguiente en la lista, si es mayor, los dos elementos cambian de lugar, esto se repite para cada elemento de la lista. En el método Bubble Sort para ordenar los datos del conjunto 7, 9, 3, 4, 8, utilizando Bubble Sort, esto significa que se intercambiarán las posiciones de los elementos para dejar al principio los valores más pequeños, por ejemplo tomando los valores 7 y 9, los

tenemos que ubicar de forma que el número 7 se encuentre a la izquierda del 9, así llegando al resultado 3, 7, 9, 4, 8.

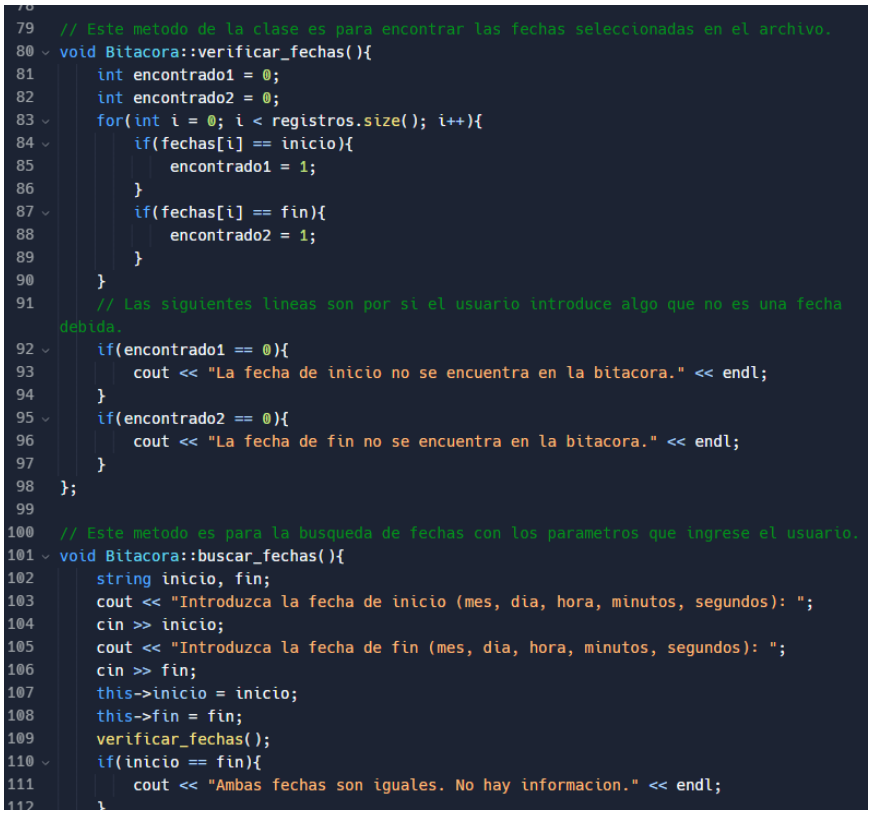
Parte 3: Solicitando al usuario un intervalo de búsqueda

En esta parte, fue necesario crear una función para que el usuario introduzca 2 fechas de inicio y fin de la búsqueda. Por ejemplo, del 16 de mayo del 2016 al 23 de junio del 2018, y nuestro programa solo imprime los elementos de la lista dentro de ese intervalo. Para esto fue necesario hacer varias cosas, comenzando con crear dos strings de inicio y fin en nuestra bitácora.



```
13      string inicio, fin;
```

Después de eso, debimos de escribir una forma para que el programa identifique los elementos de la lista ya ordenada y excluye los elementos de esta en la impresión. Para solicitar al usuario un intervalo de búsqueda, el código debe contener una función de búsqueda que lea la entrada del usuario y luego use esa información para realizar la búsqueda. En el código proporcionado, esta función se encuentra en la clase Bitácora y se llama buscar_fechas(). Esta función lee los datos de inicio y fin del usuario con los comandos cin >> inicio y cin >> fin, respectivamente. Luego, verifica si las fechas introducidas están en el archivo de bitácora y, de ser así, imprime los registros entre esos dos límites.



```
// Este metodo de la clase es para encontrar las fechas seleccionadas en el archivo.
void Bitacora::verificar_fechas(){
    int encontrado1 = 0;
    int encontrado2 = 0;
    for(int i = 0; i < registros.size(); i++){
        if(fechas[i] == inicio){
            encontrado1 = 1;
        }
        if(fechas[i] == fin){
            encontrado2 = 1;
        }
    }
    // Las siguientes lineas son por si el usuario introduce algo que no es una fecha debida.
    if(encontrado1 == 0){
        cout << "La fecha de inicio no se encuentra en la bitacora." << endl;
    }
    if(encontrado2 == 0){
        cout << "La fecha de fin no se encuentra en la bitacora." << endl;
    }
};

// Este metodo es para la busqueda de fechas con los parametros que ingrese el usuario.
void Bitacora::buscar_fechas(){
    string inicio, fin;
    cout << "Introduzca la fecha de inicio (mes, dia, hora, minutos, segundos): ";
    cin >> inicio;
    cout << "Introduzca la fecha de fin (mes, dia, hora, minutos, segundos): ";
    cin >> fin;
    this->inicio = inicio;
    this->fin = fin;
    verificar_fechas();
    if(inicio == fin){
        cout << "Ambas fechas son iguales. No hay informacion." << endl;
```

Parte 4: Imprimiendo la lista ordenada en un nuevo archivo .txt

En esta última sección, lo que tenemos que hacer es imprimir los nuevos resultados de nuestro algoritmo ordenado en un nuevo archvo.txt, al que nombraremos bitacora_ordenada. En este archivo se imprimirán incluso si el usuario definió los parámetros de qué fechas se iban a seleccionar.

```
void Bitacora::imprimirN(){
    ofstream archivo;
    archivo.open("bitacora-2.txt");
    for(int i = 0; i < registros.size(); i++){
        archivo << registros[i] << endl;
    }
    archivo.close();
};
```

Este código abre un archivo de nombre "bitacora-2.txt" en modo escritura y recorre un vector llamado registros, escribiendo cada elemento del vector en el archivo. Al terminar de recorrer el vector, el archivo es cerrado.

La impresión de la lista ordenada en un nuevo .txt se realiza mediante la función "imprimir ()" de la clase "Bitacora". Esta función recorre el vector de registros y los imprime uno por uno en un nuevo archivo .txt. El código lee un archivo .txt de entrada llamado "bitacora-1.txt" y luego se utilizan algoritmos de ordenamiento como Bubble Sort y Insertion Sort para ordenar los registros. La función "verificar_fechas ()" comprueba si las fechas de inicio y fin se encuentran en el archivo .txt de entrada y la función "buscar_fechas ()" devuelve los registros que se encuentran entre las fechas de inicio y fin. Finalmente, la función "imprimir ()" imprime los registros ordenados en un nuevo archivo .txt.

```
136 // Al final esta funcion debe de imprimir los resultados de la bitacora en un archivo
    .txt
137 void Bitacora::imprimir(){
138     for(int i = 0; i < registros.size(); i++){
139         cout << registros[i] << endl;
140     }
141 };
142
143 //Lo que hace el siguiente código es escribir un archivo .txt de los registros que
    fueron almacenados en el vector.
144 //Para eso creamos el archivo de texto llamado bitacora-2.txt asi analizando el
    registro de datos con un ciclo for y despues imprimiendo los registros en el nuevo
    archivo.
145 void Bitacora::imprimirN(){
146     ofstream archivo;
147     archivo.open("bitacora-2.txt");
148     for(int i = 0; i < registros.size(); i++){
149         archivo << registros[i] << endl;
150     }
151     archivo.close();
```

Investigación y reflexión | Arturo

Los algoritmos de ordenamiento y búsqueda son una herramienta esencial para la solución de problemas en computación: Estos algoritmos nos ayudan a encontrar una solución óptima para un problema dado, es decir, encontrar la manera más rápida y eficiente de resolverlo.

En el caso de los algoritmos de ordenamiento, estos nos ayudan a ordenar un conjunto de datos según una cierta regla, como por ejemplo de menor a mayor o de mayor a menor. Los algoritmos de ordenamiento evaluados para este caso fueron el Insertion Sort y Bubble Sort.

El Insertion Sort es un algoritmo de ordenamiento que comienza desde una posición determinada en una lista de datos y luego desplaza los elementos de la lista a la posición adecuada. Esto significa que el Insertion Sort es un algoritmo de ordenamiento que se encuentra entre los más eficientes, ya que su complejidad temporal es O(n^2). Esto quiere decir que su eficiencia se mantiene constante a medida que la lista de datos aumenta.

Por otro lado, el Bubble Sort es un algoritmo de ordenamiento que consiste en comparar dos elementos de la lista de datos y luego intercambiarlos de posición si no están ordenados de acuerdo a la regla establecida. A diferencia del Insertion Sort, el Bubble Sort tiene una complejidad temporal de O(n^2) y su eficiencia se reduce a medida que la lista de datos aumenta.

Por lo tanto, si los datos son grandes y la velocidad es importante podemos depender un poco más de Insertion Sort. Si los datos son pequeños o si la implementación es más importante que la velocidad, Bubble Sort es la mejor opción.

Investigación y reflexión | Grant

Los algoritmos de ordenamiento son de los más importantes que puede utilizar un programador, ya que se utilizan bastante en todos lados. Pero tenemos varias opciones para utilizar estos algoritmos, y en diferentes casos debemos utilizar varios algoritmos para diferentes objetivos.

En un video, aprendí las diferentes ventajas y desventajas que tienen distintos algoritmos de ordenamiento. Algunos funcionan mejor en listas de datos más grandes. Una de las formas más efectivas que nosotros como programadores podemos elegir el mejor algoritmo de ordenamiento, es fijándonos en la complejidad de este. Usualmente los que tienen una de O(n^2) o mayor son menos efectivos cuando se lidiar con listas grandes tales como la de esta tarea.

Esta es la

Notas y Referencias

[1] Sorting Algorithms Explained Visually: https://www.youtube.com/watch?v=RfXt_qHDEPw

[2] R. (2020, 4 enero). *Bubble sort*. Include Poetry. https://www.include-poetry.com/Code/C++/Metodos/Ordenamientos/Bubble-sort/

[3]*Conociendo el ordenamiento por inserción (Insertion Sort)* | . (s. f.-b). Binary Coffee. https://binarycoffee.dev/post/insertion-sort