Tecnológico de Monterrey

Actividad 1.3

Programación de Estructura de Datos y Algoritmos Fundamentales

Profesor:
Baldomero Olvera V.

Andrés Martínez - A00227463

Instrucciones

¿Qué tengo que hacer?

El canal de Suez es un canal navegable que conecta el mar Mediterráneo con el mar Rojo a través de alrededor de 190 km, uniendo por un el lado del mar Mediterráneo desde el puerto Said hasta el golfo de Suez en el mar Rojo. Este canal es navegado entre 49 y 97 barcos diariamente. Se tiene un registro de los barcos que navegan por el canal con el siguiente formato:

```
<fecha> <hora> <punto-entrada> <UBI-Identificador único del buque>
```

Ejemplo:

```
3-jan-20 13:45 M 8PAK7
```

Donde el punto de entrada puede ser ***M – Mar Mediterráneo*** y ***R – Mar Rojo***.

En equipos de tres personas, hacer una aplicación que:

1. Solicite el nombre del archivo de entrada (Ej. *bitácora.txt*
1. Download bitácora.txt) y lo abra, almacenando los datos en un vector.
2. Ordene la información por UBI + Fecha (primero por UBI, al haber empate ordenar por fecha).
3. Solicite al usuario la serie a buscar (los primeros tres caracteres de el UBI).
4. Despliegue todas las entradas al canal de los buques de esas series en forma ordenada UBI+Fecha.

Realizar una investigación y reflexión en forma individual de la importancia y eficiencia del uso de los diferentes algoritmos de ordenamiento y búsqueda en una situación problema de esta naturaleza, generando un documento llamado "**ReflexAct1.3.pdf**"

Código

Algoritmo de Búsqueda Binario

```
void busqBinaria(std::vector<std::string> a, std::string clave){
    if (clave.length() != 3) {
      std::cout << "Por favor elige una clave de 3 digitos." << std::endl;
      return;
    }
    int bajo = 0;
    int alto = a.size()-1;

    while(bajo<=alto){
        int mitad = (alto+bajo)/2;
        if(get_id(a[mitad]).substr(0,3) == clave){
            int i = 1;
            int j = 1;
            int inicio = 0;
            int fin = 0;
```

```
            while(get_id(a[mitad-i]).substr(0,3) == clave){
                inicio = mitad-i;
                i++;
            }
            while(get_id(a[mitad+j]).substr(0,3) == clave){
                fin = mitad+j;
                j++;
            }

            for(int k = inicio; k <= fin; k++){
                std::cout << a[k].substr(0,24) << " en posicion " << k+1 <<
std::endl;
            }
            break;

        } else if(clave < get_id(a[mitad]).substr(0,3)){
            alto = mitad - 1;
        } else{
            bajo = mitad + 1;
        }
    }
}
```

Algoritmo de Ordenamiento Merge

```
void une(std::vector<std::string>& a, int inicio, int mitad, int
fin){
    int i  = inicio;
    int j = mitad+1;
    int k = inicio;
    std::vector<std::string> aux(a.size());

    while(i<=mitad && j<=fin){
        if(get_id(a[i]) < get_id(a[j])){
            aux[k] = a[i];
            i++;
        } else if(get_id(a[i]) == get_id(a[j])){    //UBI es igual
            std::string fecha1 = a[i].substr(6,4) +
a[i].substr(3,2) + a[i].substr(0,2);
            std::string fecha2 = a[j].substr(6,4) +
a[j].substr(3,2) + a[j].substr(0,2);

            if(fecha1<=fecha2){
                aux[k] = a[i];
                i++;
            } else{
                aux[k] = a[j];
                j++;
            }

        } else{
            aux[k] = a[j];
            j++;
        }
        k++;
```

```
    }

    while(i<=mitad){
        aux[k] = a[i];
        i++;
        k++;
    }

    while(j<=fin){
        aux[k] = a[j];
        j++;
        k++;
    }

    for(int s=inicio; s<=fin; s++){
        a[s] = aux[s];
    }
}

void merge(std::vector<std::string>& a, int inicio, int fin){
    if(inicio<fin){
        int mitad = (inicio+fin)/2;
        merge(a, inicio, mitad);
        merge(a, mitad+1, fin);
        une(a, inicio, mitad, fin);
    }
}
```

Investigación

Antes de realizar esta actividad, se investigó sobre los diferentes tipos de algoritmos de ordenamiento y búsqueda que existen para así encontrar dos que ayudaran a tanto el ordenamiento como la búsqueda de datos en el archivo. A continuación se explicarán los algoritmos de ordenamiento y búsqueda que se consideraron para la realización de la actividad pero que no fueron parte del resultado final, estos últimos se explicarán en la sección posterior.

- Algoritmos de Ordenamiento
    - Algoritmo de Ordenamiento Burbuja
        - Este método se basa en comparar todos los elementos entre todos para así ir moviendo poco a poco los elementos más chicos a la izquierda y así terminar con una lista ordenada.
        - Este algoritmo (de complejidad $O(n^2)$ al tener que checar todos los elementos unos con otros dependiendo del número de elementos) no se eligió para la actividad debido a su rendimiento, sin embargo sí se consideró y se hicieron diferentes pruebas principalmente por su facilidad de implementación como una solución temporal.
    - Algoritmo de Ordenamiento de Selección
        - Este método consiste en buscar el elemento más chico de todos los elementos en un vector, y moverlo hasta la primera posición, de ahí seguir buscando hasta el siguiente valor más chico y moverlo a la izquierda y así sucesivamente hasta tener una lista totalmente ordenada de izquierda a derecha, asumiendo que se ordena de manera ascendente.
        - Este algoritmo no solo no se implementó por su complejidad (siendo de $O(n^2)$ un nivel mayor al elegido posteriormente) sino que su algoritmo resultaba un poco más difícil de comprender a simple vista que el algoritmo de ordenamiento burbuja, de una complejidad similar.
    - Algoritmo de Ordenamiento de Inserción
        - En el algoritmo de inserción se checan los elementos comenzando por la izquierda, y dependiendo del mismo se van comparando los distintos elementos (el primero con el segundo, el segundo con el tercero y así sucesivamente) haciendo múltiples iteraciones hasta finalmente obtener vector o arreglo ordenado.
        - Las razones por las que no se implementó este algoritmo son bastante similares a las anteriores, principalmente por su complejidad de $O(n^2)$ y por ser un poco menos legible a simple vista en comparación al algoritmo de ordenamiento burbuja, de la misma complejidad.
    - Algoritmo de Ordenamiento Rápido (Quick Sort)
        - Finalmente, el algoritmo de ordenamiento rápido (Quicksort) consiste en elegir un elemento en el medio del vector (también conocido como pivote o elemento elegido) e ir dividiendo los elementos dependiendo de si el elementos es mayor o menor del pivote, de ahí se van eligiendo diferentes pivotes y así sucesivamente hasta lograr una lista completamente ordenada.
        - Este algoritmo, a pesar de que tenga una complejidad similar al utilizado con $O(n * log\, n)$, tiene la desventaja de que en algunos casos posee una complejidad $O(n^2)$, principalmente porque depende del número de elementos

a comparar y de si estos ya están ordenados o no, por encima de si el algoritmo solamente los checa o no (como pasa con el algoritmo escogido)

- Algoritmo de Búsqueda
    - Algoritmo de Búsqueda Lineal
        - El principal funcionamiento de este algoritmo es, al darle un vector / arreglo de cierto número de elementos (n) así como un valor a buscar, este recorrerá la lista yendo elemento por elemento, iniciando por el primero, hasta encontrar el valor buscado inicialmente.
        - A pesar de que originalmente se planteó utilizar este algoritmo, su principal problema residía en su complejidad $O(n)$, una complejidad peor a la obtenida por el algoritmo de búsqueda binaria.

Complejidad de los Algoritmos Utilizados

- Algoritmo de Ordenamiento:
    - Algoritmo de Ordenamiento Merge:
        - Este algoritmo, de manera similar al algoritmo de búsqueda binaria, divide su vector en diferentes partes hasta tal punto que puede comparar de 2 en 2 elementos. Después, dichas comparaciones se hacen con los elementos adyacentes (y proceden a reunificar) hasta llegar a una unificación total del vector.
        - Para este caso, no solo se utilizó los identificadores UBI, sino que también se tomaron en cuenta las fechas y se realizaron diferentes comparaciones para ordenar dichos elementos de forma cronológica en caso de que los identificadores fueran idénticos.
        - Por lo mismo que tanto la división como la "reunificación" de elementos del vector ordenado depende enteramente del número de elementos del mismo, podemos considerar su complejidad como $O(n * log\, n)$. Este tipo de complejidad, por lo mismo que depende única y exclusivamente del número de elementos (y no de si está ordenado o no), incluso en el peor de los casos sería de $O(n\ *\ log\, n)$
- Algoritmo de Búsqueda:
    - Búsqueda Binaria:
        - Para este algoritmo, asumiendo que tenemos un vector ordenado, elegimos un punto medio y a partir de si el elemento es mayor o menor al punto en el medio elegido, decidimos elegir un punto medio de la sección que acabamos de dividir, y así sucesivamente hasta encontrar nuestro valor solicitado. Para el caso de esta actividad, decidimos imprimir los vectores de manera lineal una vez obtenida esta serie de líneas que coincidan con los primeros 3 caracteres del UBI.
        - Por lo mismo que el número de divisiones depende completamente del número de elementos del vector (ya que el que el vector *debe* de estar ordenado antes de realizar este algoritmo), la complejidad de este algoritmo es de $O(log\, n)$.

Reflexión

Son muchos los elementos que influyen en el desarrollo de software, desde la toma de decisiones sobre qué se hará hasta la tecnología que se implementará y sus utilidades. Sin embargo, también debemos de considerar la complejidad y la eficiencia de los algoritmos que utilizamos para realizar dichos programas, ya que pueden tanto elevar la popularidad y el alcance de un producto como taparlo y dejarlo en la irrelevancia.

A partir de esta actividad hemos podido comprender la importancia de elegir bien los algoritmos de búsqueda y ordenamiento para la resolución de este problema. En el caso de los algoritmos de ordenamiento, a pesar de que si bien podríamos haber utilizado un algoritmo de ordenamiento burbuja, algoritmo que originalmente se planteó para la resolución de esta actividad, nosotros consideramos que este mismo no sería el más eficiente y que habría mejores opciones para realizar dicha tarea, especialmente para un número tan alto de valores a comparar.

Para la realización de esta actividad, nosotros decidimos comparar los diferentes algoritmos de ordenamiento y búsqueda no solo basándonos en su complejidad, sino también en su legibilidad y facilidad para ser leídos y entendido, cosa que también afecta mucho en el desarrollo de software y que, en ciertos casos, también puede impactar el tiempo de desarrollo de software.

Finalmente, hemos comprendido la importancia de la complejidad en casos reales, y de cómo los diferentes algoritmos de ordenamiento y búsqueda pueden impactar en gran medida a la resolución de problemas de software y desarrollo.