RelfexActIntegradora1

Marcelo Hernandez A01194283

Para esta actividad integradora he utilizado algoritmos de búsqueda y de ordenamiento. Para la actividad se nos entrega una bitácora de información. La cual tiene aproximadamente 16,000 entradas. Después se desarrolló un código que convierte estas entradas en objetos, crea un documento ordenado entorno a la fecha, y después arroja la información de las entradas entre dos fechas determinadas por un usuario. Para tener un numero que acomodar, y en el cual poder buscar las fechas, decidí cambiarlas a segundos.

Debido a que al principio la información no esta ordenada, el primer algoritmo de búsqueda es linear. Pero este es con dos loops, uno adentro del otro. Esto porque al momento de codificar intente sortear directamente mis objetos, pero debido a falta de conocimiento sobre objetos mi código no funcionaba. Por lo cual sortee un arreglo con la información de la fecha y luego en base a un algoritmo de búsqueda lineal, imprimí los objetos ordenados en otro documento. Como ‘y’ y ‘w’ tienen exactamente el mismo tamaño, la notación big O de este algoritmo es de O(n^2). Por lo cual no es muy eficiente, pero fue de la única manera que logre que funcionara.

```
ofstream archivoSorted("bitacoraSorted.txt");
for (int y = 0; y < cantidadDeLineas; y++){
  for (int w = 0; w < cantidadDeLineas; w++){
    if (arr[y] == b1[w].getFechaSegundos()){
      archivoSorted<<b1[w].getInfo()<<'\n';
    }
  }
}
archivoSorted.close();
```

Después de decidir que no iba a acomodar los objetos directamente, sino un arreglo con su información, me di cuenta de que la eficiencia de mi algoritmo se iba a comprometer con la búsqueda linear. Por lo cual no iba a ser necesario usar un algoritmo de ordenamiento mas eficiente que este mismo. Por esto, decidí usar insertion sort.

```
void insertionSort(int *arr, int cantidadDeLineas, Bitacora *b1){//ordenar
el arreglo
  int j = 0;
  int key = 0;

  for(int i = 1; i< cantidadDeLineas; i++){
    key = arr[i];
    j = i - 1;

    while(j>= 0 && arr[j] > key){
      arr[j + 1] = arr[j];
      j = j - 1;
    }

    arr[j + 1] = key;

  }
}
```

Este tiene la misma eficiencia que el código de búsqueda linear.

Después para buscar las fechas ingresadas por el usuario utilice un binary search.

```
int binarySearch(int *arr, int *arrBinary, int cantidadBuscada){
  //buscar la posicion de las fechas que ingreso el usuario en el arreglo
  de fechas
  if(arr[arrBinary[1]] == cantidadBuscada){
    return arrBinary[1];
  }
  if (arrBinary[0] == arrBinary[1] || arrBinary[1] == arrBinary[2]){
    return arrBinary[1];
  }
  else{
    if(arr[arrBinary[1]] > cantidadBuscada){

    arrBinary[2] = arrBinary[1];
    arrBinary[1] = (arrBinary[0] + arrBinary[2]) / 2;
    return binarySearch(arr, arrBinary, cantidadBuscada);
    }

    if(arr[arrBinary[1]] < cantidadBuscada){

    arrBinary[0] = arrBinary[1];
    arrBinary[1] = (arrBinary[0] + arrBinary[2]) / 2;
    return binarySearch(arr, arrBinary, cantidadBuscada);
    }


  }
}
```

Este tiene una eficiencia big O de O(log n) la cual es mejor a la del resto de los algoritmos de mi código.

Debido a que no he cursado la materia de Objetos, mi conocimiento para poder sortearlos fue insuficiente. Aun así, resolví la problemática afrontada. En caso de que hubiera podido acomodar los objetos directamente, la búsqueda linear no hubiera sido necesaria. Por lo cual no se hubiera comprometido la eficiencia de mi algoritmo. En este caso hubiera utilizado un Merge Sort en lugar de un Insertion sort para tener una eficiencia de en el peor de los casos O(n Log n).