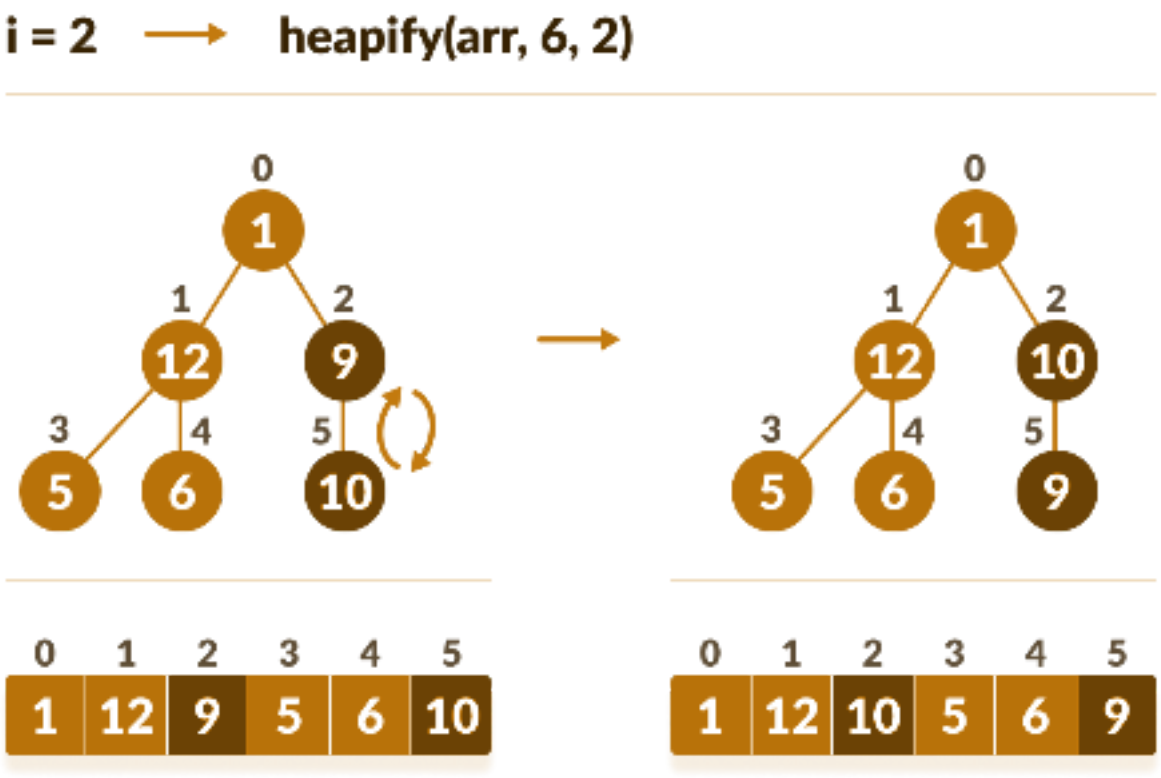
INFORME DE EVALUACIÓN N°2
ANÁLISIS DE ALGORITMOS

Proyecto de Ordenamiento



- Edgar Matus

- Daniela Galleguillos

- Oscar Peñaloza

29 de Julio de 2020

Análisis de Algoritmos

ÍNDICE

INTRODUCCIÓN

El proceso de postulación vía PSU consistía en elegir las mejores opciones de estudios superiores según los resultados PSU del estudiante. Este proyecto tiene el objetivo de ordenar y encontrar las mejores opciones de los postulantes, aplicando los algoritmos y prácticas aprendidos en el ramo de Análisis de Algoritmos UTEM.

Este programa necesita dos set de datos, admisión.csv que contiene aprox. 5M de estudiantes que son los postulantes a las diferentes carreras que ofrece la UTEM, y los códigos de las carreras con sus respectivos ponderadores, cortes de entrada, etc.

El desarrollo de este programa se encarga de ordenar los puntajes de forma descendente basado en el puntaje de corte de cada carrera, y los códigos de estas se le asignan a los archivo tipo: “código_carrera.txt” que contendrá a los postulantes en la carrera que pueden acceder, y también se cuenta con un algoritmo de búsqueda que indica donde quedó almacenado el rut del postulante.

Los métodos usados para esto fueron HeapSort y Heapify que se detallarán con más detalle a continuación.

OBJETIVOS

- 1. Investigar y usar algoritmos de ordenamiento.
- 2. Investigar y usar algoritmos de búsqueda.
- 3. Mejorar el uso de archivos.

Si bien la librería estándar de uno de los dos lenguajes requeridos y el que usamos para la tarea, contiene dos algoritmos útiles (std::sort y std::qsort) para el ordenamiento de arreglos y la clase vector, no podemos ordenar las ponderaciones sin una estructura que nos permita relacionarlas con al menos el rut de cada estudiante. En el caso de std::sort podíamos ordenar un arreglo del largo del n° de estudiantes con un rendimiento bastante bueno, pero requerimos de relacionar a cada estudiante con su rut y se debe ir a la implementación de la librería estándar para extraer el algoritmo y editarlo de manera de hacer esto posible. Lo anterior es, a priori, innecesario debido a la existencia de std::qsort que además de poder ordenar arreglos bidimensionales, nos permite añadir condiciones para cada columna y el orden de estas con respecto a las otras filas como lo

muestra el código a continuación:

```
#include <iostream>
#include <cstdlib>
#include <climits>

int main()
{
    int a[] = {-2, 99, 0, -743, 2, INT_MIN, 4};
    constexpr std::size_t size = sizeof a / sizeof *a;

    std::qsort(a, size, sizeof *a, [](const void* a, const void* b)
    {
        int arg1 = *static_cast<const int*>(a);
        int arg2 = *static_cast<const int*>(b);

        if(arg1 < arg2) return -1;
        if(arg1 > arg2) return 1;
        return 0;

    //  return (arg1 > arg2) - (arg1 < arg2); // possible shortcut
    //  return arg1 - arg2; // erroneous shortcut (fails if INT_MIN is present)
    });

    for(int ai : a)
        std::cout << ai << ' ';
}
```

https://en.cppreference.com/w/cpp/algorithm/qsort (Ejemplo unidimensional con un arreglo int*)

Lamentablemente el algoritmo resulta falente para arreglos bidimensionales con demasiada longitud y muy difícil de usar para tantas columnas, siendo un costo muy alto (por no decir infinito) a pagar a cambio de poder ordenar una simple matriz de tipo int.

¿Por qué elegir arreglo por sobre la clase std::vector? Trabajando con alguna de las tareas anteriormente desarrolladas por el curso pudimos ver la diferencia de trabajar con un algoritmo eficiente y útil (sin “crashes”) debido a la manipulación de tanta longitud de datos: **Heapsort y Heapify**. La diferencia de tiempo entre usar la clase y un arreglo en estas implementaciones, fue bastante significativa y teniendo la certeza de que los arreglos en C y C++ son contiguos entre sí en la memoria tenemos razones suficientes para ir por la primera opción teniendo como objetivo el lidiar con la longitud indefinida para un arreglo de punteros tipo int de nuestra potencial matriz de puntajes. La contra: es no contar con un método para “poppear” filas sin reacomodar TODOS los datos.

En adición a todo lo anterior, está el objetivo de leer, manipular y escribir archivos de texto de manera correcta y eficiente, evitando tener que re-abrirlos para re-acomodar estudiantes, surgiendo una nueva necesidad: Contar con una estructura de datos de cada carrera y ¿por qué no, de la Universidad?

ESTRUCTURAS Y FLUJOS DE LOS DATOS

Para poder crear una matriz en C y C++, no es de suma sino total importancia, saber la cantidad de filas de esta. Es por lo anterior que se requiere saber, contando las filas del archivo de texto (contar(std::istream)), la cantidad de filas y/o punteros a cada sub-arreglo con cada estudiante resultando de manera gráfica los siguiente:

int** ponderados
ponderados[0] // desde fila 1
...
ponderados[n] // por fila enésima
...
ponderados[5005715] // hasta fila 5005715

Para luego obtener con una función (obtener Puntajes(std::string)) cada fila del archivo:

int** ponderados	ponderados[n][0] // rut	ponderados[n][1] // NEM	ponderados[n][2] // ranking	ponderados[n][3] // matemática	ponderados[n][4] // lenguaje	ponderados[n][5] // ciencia	ponderados[n][6] // mejor ponderación	ponderados[n][7] // código carrera
ponderados[0]	int	int	int	int	int	int	int	int

...	int	int	int	int	int	int	int	int
ponderados[n]	int	int	int	int	int	int	int	int
...	int	int	int	int	int	int	int	int
ponderados[50 05715]	int	int	int	int	int	int	int	int

Y directamente desde uno de los headers del proyecto:

```
#ifndef ESTRUCTURAS_H
#define ESTRUCTURAS_H

#include <iostream>
#include <string>


typedef struct carrera{
    const char* nombre;
    int codigo;
    float ponderaciones[5];
    int vacantes;
    double primero;
    double ultimo;
    int** mechones;
    bool disponibilidad;
};

typedef struct universidad{
    const char* nombre;
    carrera* oferta;
    int carreras;
    bool disponibilidad;
};


#endif /* ESTRUCTURAS_H */

```

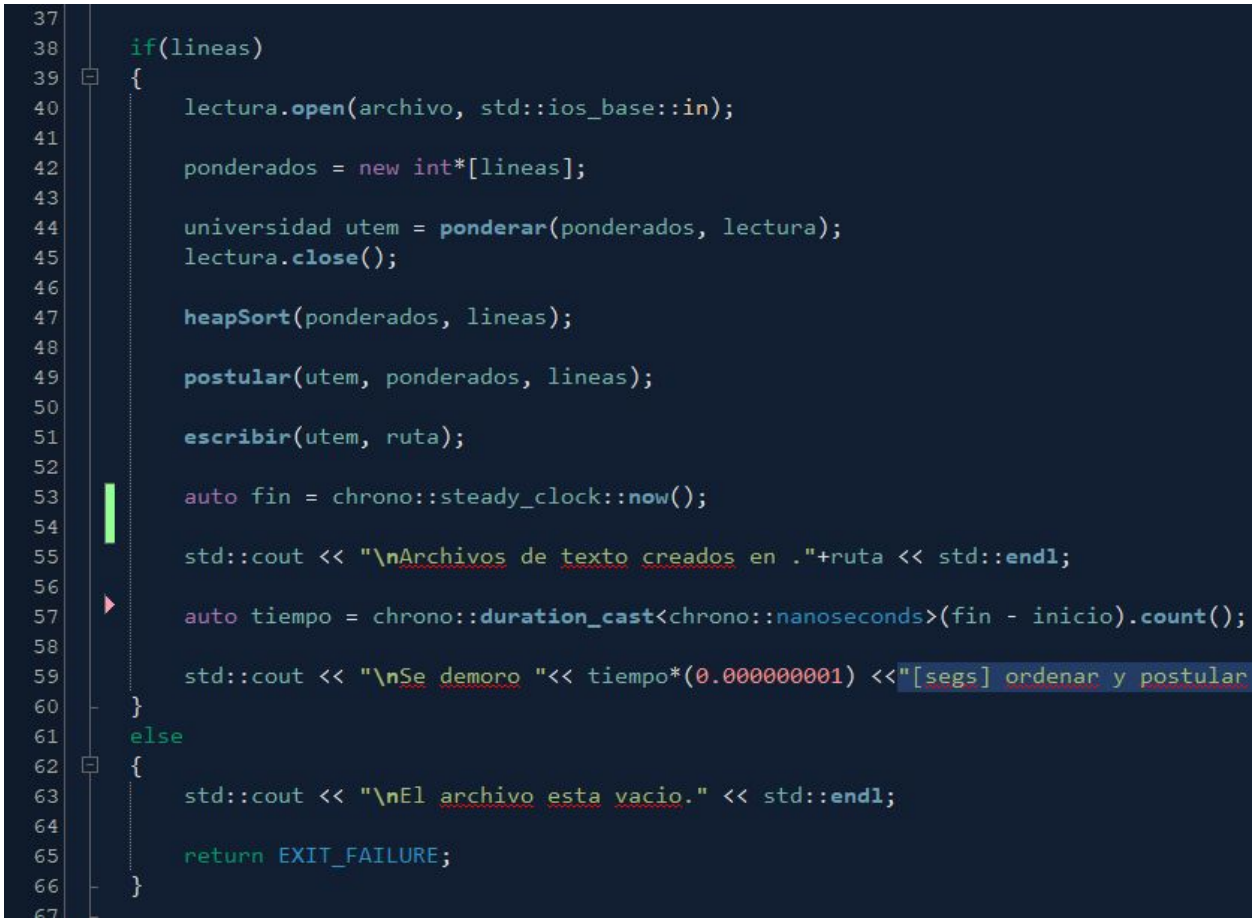
Estas dos son las estructuras declaradas dentro del proyecto para almacenar a los estudiantes ideales que rindieron la prueba de selección universitaria.

Donde la función ponderar toma la matriz y el contenido del archivo para ponderar a cada estudiante en su mejor opción. Pero sin antes de inicializar todas las carreras y sus

respectivos atributos para así retornar la estructura universidad (definida como dato). Una vez la matriz obtiene para cada estudiante su código de carrera esta es enviada a la implementación del heapSort para ordenarla según el puntaje de cada mejor opción para luego poder llenar el dato de tipo universidad con la matriz de estudiantes en postular en cada carrera del arreglo oferta. Finalmente el dato de tipo universidad, utem, es pasado pasado por la función escribir(universidad,std::string) donde es recorrido con un ciclo anidado gracias a los valores de universidad.carreras (int) y carrera.vacantes (int) y se genera el output stream para cada archivo usando la conversión a string de carrera.codigo.

Para el segundo objetivo, el de buscar, también se usa un ciclo anidado pero esta vez el interior se usa el necesario para extraer cada línea del input stream, que se abre usando un arreglo de arreglo caracteres que contiene cada código con su extensión .txt en buscar.

A continuación el código del if-statement del archivo main.cpp donde ocurre este flujo:

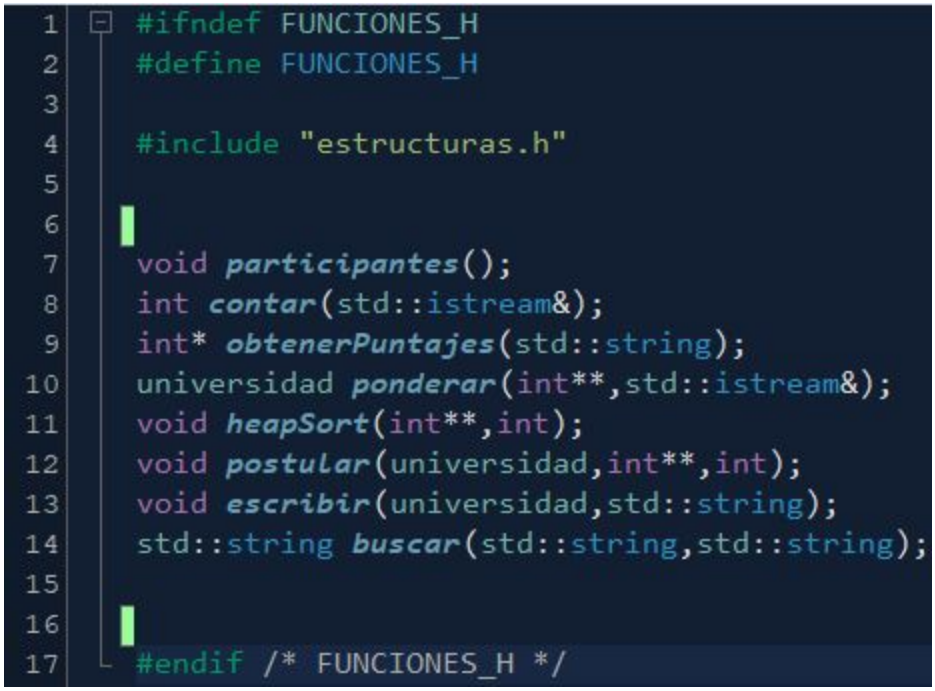


```
37
38  if(lineas)
39  {
40      lectura.open(archivo, std::ios_base::in);
41
42      ponderados = new int*[lineas];
43
44      universidad utem = ponderar(ponderados, lectura);
45      lectura.close();
46
47      heapSort(ponderados, lineas);
48
49      postular(utem, ponderados, lineas);
50
51      escribir(utem, ruta);
52
53      auto fin = chrono::steady_clock::now();
54
55      std::cout << "\nArchivos de texto creados en ."+ruta << std::endl;
56
57      auto tiempo = chrono::duration_cast<chrono::nanoseconds>(fin - inicio).count();
58
59      std::cout << "\nSe demoro "<< tiempo*(0.000000001) <<"[segs] ordenar y postular
60  }
61  else
62  {
63      std::cout << "\nEl archivo esta vacio." << std::endl;
64
65      return EXIT_FAILURE;
66  }
```

ALGORITMOS E IMPLEMENTACIONES

A continuación los algoritmos/funciones más relevantes que son encabezados con nuestro archivo principal:

0. *participantes**
1. *contar*
2. *obtenerPuntajes*
3. *ponderar*
4. *heapSort*
5. *postular*
6. *escribir*
7. *buscar*



```
#ifndef FUNCIONES_H
#define FUNCIONES_H

#include "estructuras.h"


void participantes();
int contar(std::istream&);
int* obtenerPuntajes(std::string);
universidad ponderar(int**,std::istream&);
void heapSort(int**,int);
void postular(universidad,int**,int);
void escribir(universidad,std::string);
std::string buscar(std::string,std::string);


#endif /* FUNCIONES_H */
```

(*: no cabe duda que es innecesario de explicar)

1. contar:

Es un operador bastante simple que recibe como parámetro la dirección de memoria del contenido del stream en main() y retorna la cantidad de líneas del archivo .csv (Suponiendo desde un comienzo que el archivo posee la estructura válida aún así si tiene el nombre “puntajes”).

2. obtenerPuntajes:

Es un operador que recibe el string correspondiente a cada fila, lo separa por sus respectivas “;” y convierte los subarreglos de caracteres a un int* que finalmente es retornado.

```
28  int* obtenerPuntajes(std::string fila){
29      int* arreglo = new int[8], i = 0;
30      std::stringstream ss(fila);
31      std::string item;
32
33      while (getline(ss, item, ';')) {
34          int valor = atoi(item.c_str());
35          arreglo[i] = valor;
36          i++;
37      }
38
39      arreglo[6] = 0; // historia -> mejor ponderacion
40      arreglo[7] = 0; // <- código carrera
41
42      return arreglo;
43  }
```

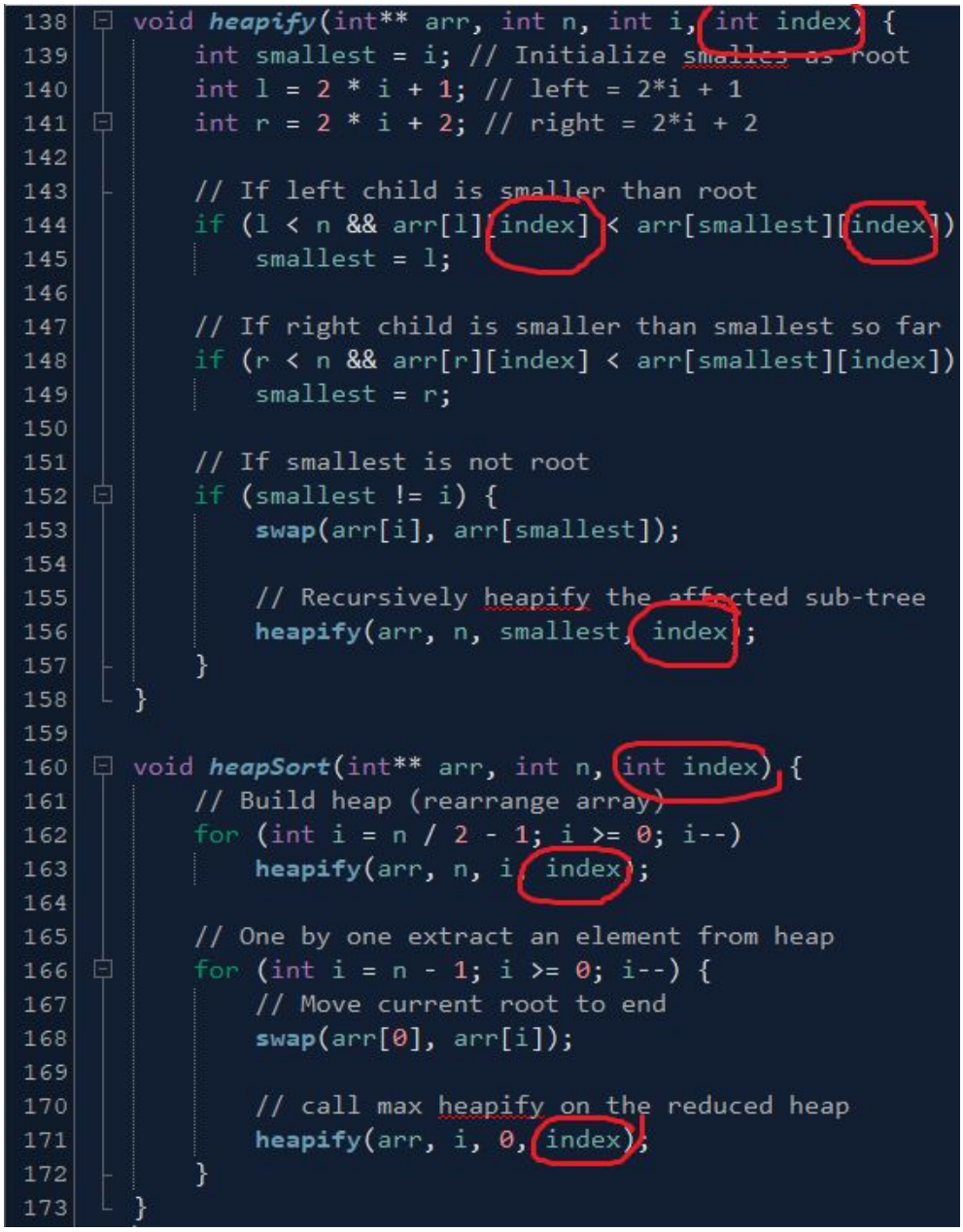
3. ponderar:

Quizá la función más grande de la documentación. Inicializa el dato de tipo universidad y recibe la potencial matriz de los puntajes de los estudiantes como un arreglo de punteros y la dirección del stream del archivo. Finalmente inicializa los arreglos universidad.mechones para cada carrera y retorna la universidad creada:

```
88      for (std::string linea; std::getline(archivo, linea); i++)
89      {
90          double mejor = 0.0;
91
92          int* puntajes = obtenerPuntajes(linea);
93
94          /*
95           *
96          ponderados[i] = new int[8];
97          ponderados[i][0] = puntajes[0]; // <- rut
98          ponderados[i][1] = puntajes[1]; // <- NEM
99          ponderados[i][2] = puntajes[2]; // <- ranking
100         ponderados[i][3] = puntajes[3]; // <- matematica
101         ponderados[i][4] = puntajes[4]; // <- lenguaje
102         ponderados[i][5] = puntajes[5]; // <- ciencias
103         ponderados[i][6] = 0; // <- mejor ponderación
104         ponderados[i][7] = 0; // <- código carrera
105         *
106         */
107
108         ponderados[i] = puntajes;
109
110         for (int j = 0; j < n; j++)
111         {
112             if ((puntajes[3]+puntajes[4]) / (double) 2.0 >= 450.0 )
113             {
114                 double ponderacion = ponderar(puntajes, U.oferta[j]);
115
116                 if (ponderacion > (double) U.oferta[j].ultimo && ponderacion > mejor)
117                 {
118                     mejor = ponderacion;
119                     ponderados[i][6] = (int)round(ponderacion);
120                     ponderados[i][7] = U.oferta[j].codigo;
121                 }
122             }
123         }
124     }
```

4. Heapsort (heapSort y heapify):

La ***crème de la crème*** y el algoritmo en torno al cual gira este proyecto. Si bien consta de dos “partes”, el nombre original es un derivado del ya conocido ***sort,*** pero introduciendo a este, la idea de crear montículos binarios de manera recursiva (valga la redundancia) para ordenar bajo algún criterio mediante a algún ***swap*** los enlazados de manera vertical unos de otros. Nuestra implementación agrega un parámetro adicional (“index”, qué nos dice la posición del elemento o columna a comparar) para poder así, operar la matriz de enteros como si fuese un arreglo unidimensional.



```
138  void heapify(int** arr, int n, int i, int index) {
139      int smallest = i; // Initialize smallest as root
140      int l = 2 * i + 1; // left = 2*i + 1
141      int r = 2 * i + 2; // right = 2*i + 2
142
143      // If left child is smaller than root
144      if (l < n && arr[l][index] < arr[smallest][index])
145          smallest = l;
146
147      // If right child is smaller than smallest so far
148      if (r < n && arr[r][index] < arr[smallest][index])
149          smallest = r;
150
151      // If smallest is not root
152      if (smallest != i) {
153          swap(arr[i], arr[smallest]);
154
155          // Recursively heapify the affected sub-tree
156          heapify(arr, n, smallest, index);
157      }
158  }
159
160  void heapSort(int** arr, int n, int index) {
161      // Build heap (rearrange array)
162      for (int i = n / 2 - 1; i >= 0; i--)
163          heapify(arr, n, i, index);
164
165      // One by one extract an element from heap
166      for (int i = n - 1; i >= 0; i--) {
167          // Move current root to end
168          swap(arr[0], arr[i]);
169
170          // call max heapify on the reduced heap
171          heapify(arr, i, 0, index);
172      }
173  }
```

5. postular:

La tarea de postular es relativamente sencilla: recibir la matriz ordenada con los puntajes y códigos de carrera respectivamente asignados en las posiciones y comenzar a llenar el arreglo U.oferta dentro de nuestra universidad en *main. **"El problema":*** debido a la naturaleza de los puntajes, las respectivas ponderaciones de cada carrera y la forma de ponderación implementada, existían casos donde muchos estudiantes recibían el mismo código de carrera y otros en donde se producía todo lo contrario, esto último no necesariamente a la ponderación de la carrera (que en muchos casos era la misma) sino que por la forma "cascada" de hacer pasar a cada estudiante conservando su mejor ponderación. Es por esto que nos vimos en la necesidad de aplicar una segunda ponderación por cada estudiante dentro de las carreras con cupo asumiendo que su segundo mejor será mayor que el de todos los demás después:

```
238     for(int i = 0; i < estudiantes; i++)
239     {
240         U.disponibilidad = cupo(U.oferta);
241
242         if(U.disponibilidad == true)
243         {
244             int* postulante = ponderados[i];
245
246             for(int j = 0; j < n; j++)
247             {
248                 if(postulante[7] == U.oferta[j].codigo)
249                 {
250                     if(U.oferta[j].disponibilidad == true)
251                     {
252                         almacenar(postulante, U.oferta, j);
253                     }
254                     else
255                     {
256                         double mejor2 = 0;
257                         int posicion;
258
259                         for(int k = 0; k < n; k++)
260                         {
261                             if(U.oferta[k].disponibilidad == true)
262                             {
263                                 double ponderacion2 = ponderar(postulante, U.oferta[k]);
264
265                                 if(ponderacion2 >= mejor2)
266                                 {
267                                     posicion = k;
268                                     mejor2 = ponderacion2;
269                                 }
270                             }
271                         }
272
273                         postulante[6] = (int)round(mejor2);
274
275                         almacenar(postulante, U.oferta, posicion);
276                     }
277                 }
278             }
279         }
```

6. escribir:

Esta funcionalidad, necesaria, cuenta con un algoritmo más que simple: declara una variable puntero para nuestro output stream usa los códigos de las carreras convertidos a string para obtener el nombre de cada archivo y al validar su apertura mediante el método de std::ofstream.

7. buscar:

La búsqueda del puntaje asociado al rut se basa en tomar uno por uno los archivos asociados a un PATH que se entrega por consola, si los caracteres del rut coinciden este despliega el rut, puntaje y nombre del archivo, que corresponde al código de la carrera.

```
    int n = sizeof(archivos) / sizeof(archivos[0]);

    for(int i = 0; i < n; i++)
    {
        lectura.open(ruta+"/"+archivos[i]);

        if(lectura.is_open())
        {

            if(existen == false)
            {
                existen = true;
            }

            int j = 0;

            for(std::string linea; std::getline(lectura, linea); j++)
            {
                int k = 0;
                std::string aux[2], item;
                std::stringstream ss(linea);

                while(getline(ss, item, ';'))
                {
                    aux[k] = item.c_str();
                    k++;
                }

                if(rut == aux[0])
                {
                    busqueda = "\nEstudiante encontrado en la linea "+std::to_string(j+1)+" del archivo "+archivos[i];

                    lectura.close();

                    break;
                }
            }
        }

        lectura.close();
    }
```

RESULTADOS

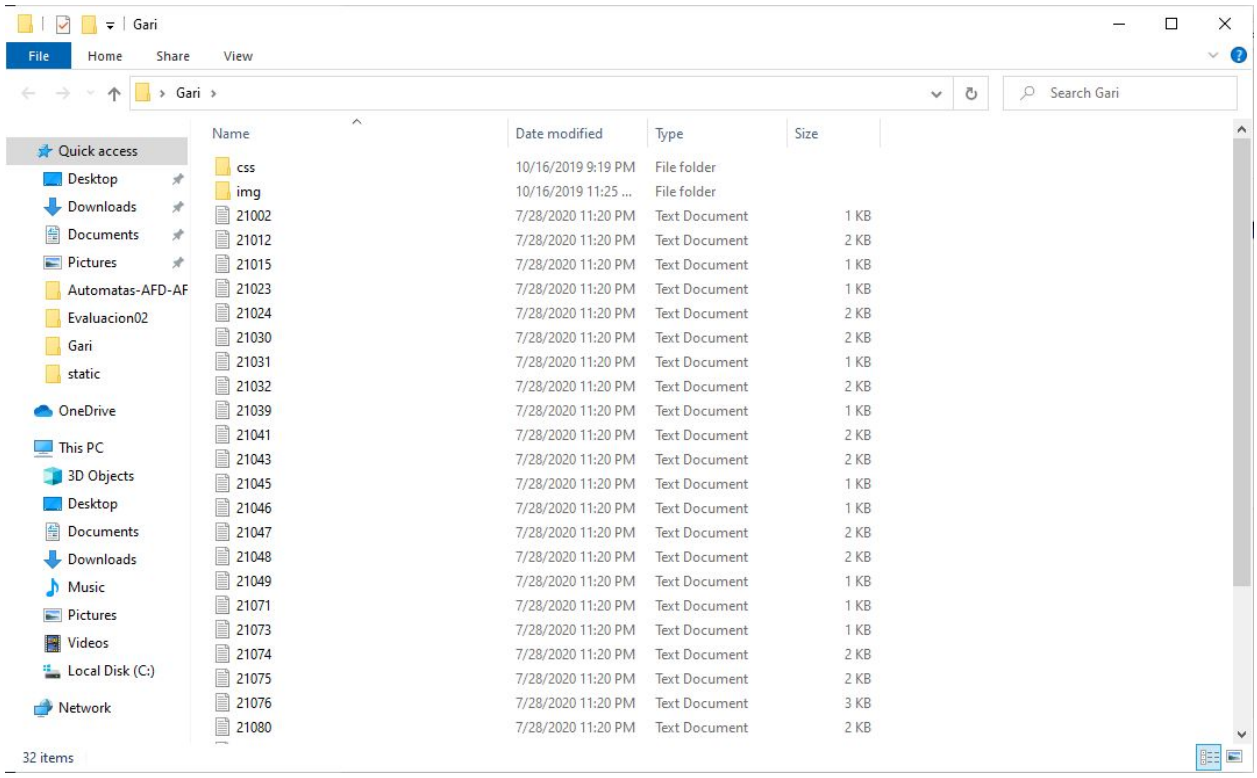
Si bien el ejecutable final consume relativamente bastantes recursos, creemos que la forma en que se afrontó puede ayudar disminuir un poco lo anterior, la cantidad de datos y la recursividad aplicada. Esto en base a los resultados de eficiencia obtenidos.

1. Entrada:

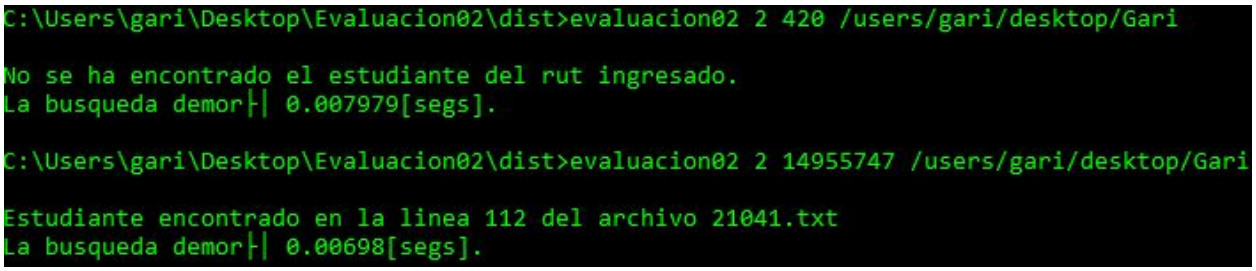
```
C:\Users\gari\Desktop\Evaluacion02\dist>evaluacion02 1 /users/gari/desktop/puntajes.csv /users/gari/desktop/Gari

Archivo encontrado.

Archivos de texto creados en ./users/gari/desktop/Gari

Se demor├| 31.1618[segs] ordenar y postular a los 5015751 estudiantes.
```

   La operación toma 31 y fracción segundos en uno de nuestros, relativamente modernos equipos. Creemos que se puede lograr una eficiencia (precisión) mejor en cuanto al objetivo que es ordenar, pero los resultados en cuanto a tiempo de ejecución son bastantes tentadores:

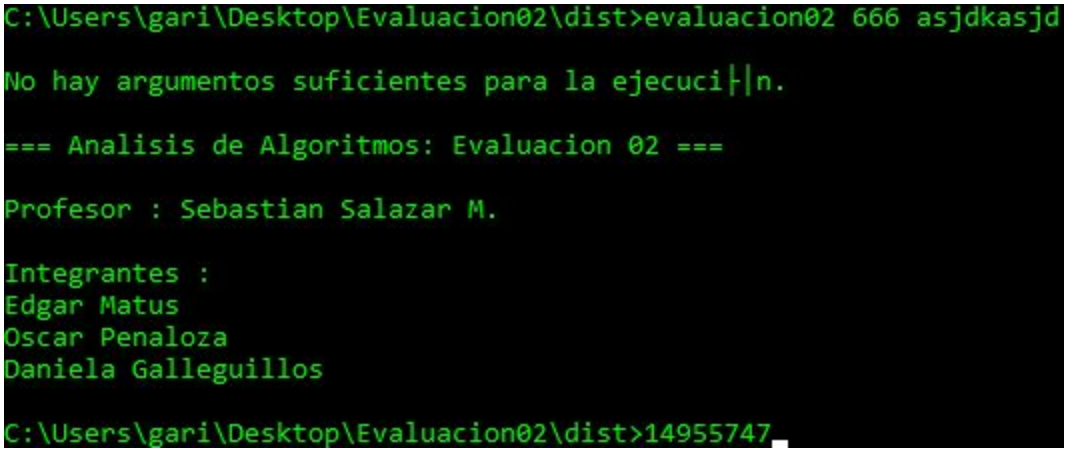
2. Salida:



3. Búsqueda:



Nuevamente, creemos necesario aclarar que aplicar algún algoritmo de búsqueda en este caso es absolutamente innecesario, debido a la capacidad de los equipos actuales, pero **sabemos** que siempre hay excepciones.



CONCLUSIONES

Debemos comenzar acotando que se cumplió con los objetivos establecidos desde un principio del trabajo. Pudiendo lograr implementar uno de los algoritmos de ordenamiento vistos en clase, cumpliendose las postulaciones de los estudiantes en las distintas carreras ofertadas.

HeapSort es un algoritmo de ordenamiento que fue elegido por la estructura del programa, ya que su implementación fue la que nos acomodó y PUDO ser implementada, debido a las necesidades presentadas durante el planteamiento inicial y desarrollo.

Con respecto a la utilización de un algoritmo lineal para la búsqueda de los estudiantes creemos que el programa funciona bien con él. Esto se debe a la rapidez de las máquinas y podemos inferir que los sistemas y arquitecturas actuales muchas veces no necesitan de otras implementaciones más complejas. Pero en el caso de una estructura más masiva, como el de una base de datos, son **indispensables** en conjunto de una formalización relacional entre los mismos datos.

Finalmente la manipulación de archivos se limitó al mínimo ya que al abrir y cerrarlos se interactúa más con eslabones más primitivos de memoria dentro de la máquina y eso se traducía sí o sí en más tiempo de ejecución.

REFERENCIAS

1. https://es.wikipedia.org/wiki/Heapsort
2. https://en.wikipedia.org/wiki/Binary_heap
3. https://en.cppreference.com/w/cpp/algorithm/qsort
4. http://www.cplusplus.com/reference/cstdlib/qsort/
5. https://www.geeksforgeeks.org/heap-sort/