**Nombre:** Isabela Zambrano **NRC:** 29852

**Fecha:** 02/02/2025

1. Se cuenta con un archivo directo llamado ALUMNOS. DAT con información de alumnos de una universidad. El archivo se abre para lectura y se indexa por medio de su clave primaria (#legajo). El índice se mantiene en memoria principal. Se ofrece la posibilidad al usuario de buscar alumnos por legajo. La búsqueda binaria se aplica sobre el índice y se accede en forma directa (seek) al archivo para recuperar el registro. Se asume que el archivo está ordenado de manera ascendente por su clave primaria (#legajo). Si éste no fuese el caso, el índice debería ordenarse antes de aplicar una búsqueda binaria sobre él. Aclaración: no se provee el archivo ALUMNOS. DAT (debido a que es binario). Deberá ser generado antes de la ejecución de este programa)

```
int busquedaBinaria(RegistroIndice* base, int cantidad, int legajoBuscado) {

    int lo = 0;
    int hi = cantidad - 1;

    while (lo <= hi) {
        int mid = (lo + hi) / 2;

        int leg = (*(base + mid)).legajo;

        if (leg == legajoBuscado)
            return mid;

        if (leg < legajoBuscado)
            lo = mid + 1;
        else
            hi = mid - 1;
    }

    return -1;
}
```

2. Localice un entero en un arreglo usando una versión recursiva del algoritmo de búsqueda binaria. El número que se debe buscar (clave) se debe ingresar como argumento por línea de comandos. El arreglo de datos debe estar previamente ordenado (requisito de la búsqueda binaria). Comparar esta versión recursiva de la búsqueda binaria con la versión iterativa dada en el capítulo 5.

```
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
// -------------------------
// BÚSQUEDA BINARIA RECURSIVA
// -------------------------
int busquedaBinariaRec(int* inicio, int* fin, int clave) {

   if (inicio > fin)
      return -1;

   int* medio = inicio + (fin - inicio) / 2;

   if (*medio == clave)
      return medio - inicio;

   if (*medio > clave)
      return busquedaBinariaRec(inicio, medio - 1, clave);

   return busquedaBinariaRec(medio + 1, fin, clave);
}

// -------------------------
// BÚSQUEDA BINARIA ITERATIVA
// -------------------------
int busquedaBinariaIter(int* inicio, int* fin, int clave) {

   int* lo = inicio;
   int* hi = fin;

   while (lo <= hi) {

      int* mid = lo + (hi - lo) / 2;

      if (*mid == clave)
         return mid - inicio;

      if (*mid < clave)
         lo = mid + 1;
      else
         hi = mid - 1;
   }

   return -1;
}

int main(int argc, char* argv[]) {

   if (argc < 2) {
      cout << "Uso: programa <clave_a_buscar>\n";
      return 1;
   }
```

```
    int clave = atoi(argv[1]);

    // Arreglo ORDENADO
    int datos[] = {1, 7, 10, 15, 20, 35, 45, 69, 71, 89};
    int n = sizeof(datos) / sizeof(int);

    int* inicio = datos;
    int* fin = datos + (n - 1);

    // ------------------
    // Búsqueda recursiva
    // ------------------
    int posRec = busquedaBinariaRec(inicio, fin, clave);

    // -------------------
    // Búsqueda iterativa
    // -------------------
    int posIter = busquedaBinariaIter(inicio, fin, clave);

    cout << "\nClave buscada: " << clave << "\n";

    if (posRec != -1)
       cout << "Recursiva: encontrado en la posicion " << posRec << "\n";
    else
       cout << "Recursiva: no encontrado.\n";

    if (posIter != -1)
       cout << "Iterativa: encontrado en la posicion " << posIter << "\n";
    else
       cout << "Iterativa: no encontrado.\n";

    return 0;
  }
```

3. Diseñe una variación de Búsqueda Binaria (algoritmo 1.4) que efectúe sólo una comparación binaria (es decir, la comparación devuelve un resultado booleano) de K con un elemento del arreglo cada vez que se invoca la función. Pueden hacerse comparaciones adicionales con variables de intervalo. Analice la corrección de su procedimiento. Sugerencia: ¿Cuándo deberá ser de igualdad (==) la única comparación que se hace?

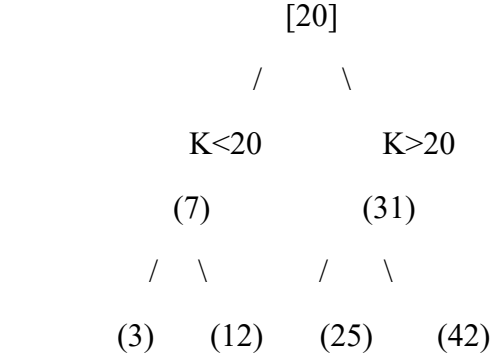
```
#include <iostream>
    using namespace std;


    int busquedaEspecial(int* E, int primero, int ultimo, int K) {
```

```
        if (ultimo < primero)
            return -1;

        int* pPrimero = E + primero;
        int* pUltimo  = E + ultimo;


        if (K < *pPrimero || K > *pUltimo)
            return -1;

        int medio = (primero + ultimo) / 2;
        int* pMedio = E + medio;

        if (K == *pMedio)
            return medio;

        if (K <= *pMedio)
            return busquedaEspecial(E, primero, medio - 1, K);
        else
            return busquedaEspecial(E, medio + 1, ultimo, K);
    }

    int main() {

        int E[] = {1, 7, 9, 14, 19, 24, 31, 42};
        int n = sizeof(E) / sizeof(E[0]);

        int K;
        cout << "Ingrese el valor a buscar: ";
        cin >> K;

        int indice = busquedaEspecial(E, 0, n - 1, K);

        if (indice == -1)
            cout << "No encontrado\n";
        else
            cout << "Encontrado en posicion: " << indice << "\n";

        return 0;
}
```

4. ¿Cómo podría modificar Búsqueda Binaria (¿algoritmo 1.4) para eliminar trabajo innecesario si se tiene la certeza de que K está en el arreglo? Dibuje un árbol de decisión para el algoritmo modificado con n = 7. Efectúe análisis de comportamiento promedio y de peor caso. (Para el promedio, puede suponerse que n = 2 - 1 para alguna k.)

```
#include <iostream>
using namespace std;

int busquedaAsumida(int* A, int n, int K) {
```

```
    int low = 0;
    int high = n - 1;

    while (true) {
        int mid = (low + high) / 2;
        int valor = *(A + mid);

        if (valor == K)
            return mid;

        if (valor < K)
            low = mid + 1;
        else
            high = mid - 1;
    }
}

int main() {
    int datos[] = {3, 7, 12, 20, 25, 31, 42};
    int n = sizeof(datos) / sizeof(datos[0]);

    int K;
    cout << "Ingrese un valor que SI esté en el arreglo: ";
    cin >> K;

    int pos = busquedaAsumida(datos, n, K);

    cout << "Encontrado en la posicion: " << pos << endl;
    cout << "Valor verificado: " << *(datos + pos) << endl;

    return 0;
}
```




5. Encontrar el algoritmo de búsqueda binaria para encontrar un elemento K en una lista de elementos $X_1$, $X_2$, ..., $X_n$, previamente clasificados en orden ascendente.

El array o vector X se supone ordenado en orden creciente si los datos son numéricos, o alfabéticamente si son caracteres. Las variables BAJO, CENTRAL, ALTO indican los límites inferior, central y superior del intervalo de búsqueda.

```
algoritmo busqueda_binaria
//declaraciones
inicio
 //1lenar (X, N)
//ordenar (X, N)
leer(K)
//inicializar variables
BAJO← 1
ALTO← N
CENTRAL ← ent ((BAJO + ALTO) / 2)
 mientras (BAJO < ALTO) y (X[CENTRAL] <> K) hacer
si K<X[CENTRAL] entonces
ALTO ←CENTRAL-1
si_no
BAJO ← CENTRAL+1
fin_si
CENTRAL←ent ((BAJO + ALTO) / 2)
fin_mientras
si K = X[CENTRAL] entonces
escribir('Valor encontrado en', CENTRAL)
si_no
escribir('Valor no encontrado')
fin_si
fin
```

**Pseudocódigo corregido**

```
algoritmo BUSQUEDA_BINARIA
   // X es un arreglo ordenado de tamaño N
   leer(K)

   BAJO ← 1
   ALTO ← N
   CENTRAL ← truncar( (BAJO + ALTO) / 2 )

   mientras BAJO ≤ ALTO y X[CENTRAL] ≠ K hacer
      si K < X[CENTRAL] entonces
         ALTO ← CENTRAL - 1
      si_no
         BAJO ← CENTRAL + 1
```

```
        fin_si

        CENTRAL ← truncar( (BAJO + ALTO) / 2 )
    fin_mientras

    si X[CENTRAL] = K entonces
        escribir("Valor encontrado en ", CENTRAL)
    si_no
        escribir("Valor no encontrado")
    fin_si
fin_algoritmo
```

```
#include <iostream>
using namespace std;

int busquedaBinaria(int* X, int N, int K) {
    int BAJO = 0;
    int ALTO = N - 1;
    int CENTRAL = (BAJO + ALTO) / 2;

    // mientras BAJO <= ALTO y *(X + CENTRAL) != K
    while (BAJO <= ALTO && *(X + CENTRAL) != K) {

        if (K < *(X + CENTRAL)) {
            ALTO = CENTRAL - 1;
        } else {
            BAJO = CENTRAL + 1;
        }

        CENTRAL = (BAJO + ALTO) / 2;
    }

    if (BAJO <= ALTO && *(X + CENTRAL) == K)
        return CENTRAL;
    else
        return -1;
}

int main() {
    int X[] = {3, 8, 12, 17, 25, 30, 45};
    int N = sizeof(X) / sizeof(X[0]);

    int K;
    cout << "Ingrese el valor a buscar: ";
    cin >> K;

    int pos = busquedaBinaria(X, N, K);
```

```
    if (pos != -1)
        cout << "Valor encontrado en la posicion " << pos << endl;
    else
        cout << "Valor no encontrado" << endl;

    return 0;
}
```

6. Se cuenta con una lista de Pokémons, de cada uno de estos se sabe su nombre, número y tipo (solo considerar uno el principal), con los cuales deberá resolver las siguientes tareas: determinar si existe el Pokémon Cobalion y mostrar toda su información.

```
#include <iostream>
#include <cstring>
using namespace std;

class Pokemon {
private:
    char nombre[50];
    int numero;
    char tipo[30];

public:
    Pokemon() {
        nombre[0] = '\0';
        tipo[0] = '\0';
        numero = 0;
    }

    void ingresar() {
        cout << "Nombre: ";
        cin >> nombre;
        cout << "Numero: ";
        cin >> numero;
        cout << "Tipo: ";
        cin >> tipo;
    }

    const char* getNombre() { return nombre; }
    int getNumero() { return numero; }
    const char* getTipo() { return tipo; }

    void mostrar() {
        cout << "---------------------------------\n";
        cout << "Nombre : " << nombre << "\n";
        cout << "Numero : " << numero << "\n";
        cout << "Tipo   : " << tipo << "\n";
```

```
        cout << "---------------------------------\n";
    }
};

class ListaPokemons {
private:
    Pokemon lista[100];
    int tam;

    // Ordenar alfabéticamente por nombre (burbuja simple)
    void ordenar() {
        for (int i = 0; i < tam - 1; i++) {
            for (int j = i + 1; j < tam; j++) {
                if (strcmp(lista[i].getNombre(), lista[j].getNombre()) > 0) {
                    Pokemon temp = lista[i];
                    lista[i] = lista[j];
                    lista[j] = temp;
                }
            }
        }
    }

public:
    ListaPokemons() { tam = 0; }

    void ingresarPokemons(int n) {
        for (int i = 0; i < n; i++) {
            cout << "\nIngresando Pokemon #" << (i + 1) << "\n";
            lista[i].ingresar();
        }
        tam = n;
        ordenar(); // Ordenar después de ingresar
    }

    // Búsqueda binaria por nombre
    int buscarCobalion() {
        int low = 0;
        int high = tam - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            int cmp = strcmp(lista[mid].getNombre(), "Cobalion");
            if (cmp == 0)
                return mid; // encontrado
            else if (cmp < 0)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1; // no encontrado
    }
```

```
    void mostrarCobalion() {
        int pos = buscarCobalion();
        if (pos != -1) {
            cout << "\n*** Cobalion ENCONTRADO ***\n";
            lista[pos].mostrar();
        } else {
            cout << "\nCobalion NO se encuentra en la lista.\n";
        }
    }
};

int main() {
    ListaPokemons pokedex;

    int n;
    cout << "Cuantos Pokemons desea ingresar? ";
    cin >> n;

    pokedex.ingresarPokemons(n);
    pokedex.mostrarCobalion();

    return 0;
}
```

7. Se dispone de la lista de superhéroes y villanos de la saga de Marvel Cinematic Universe (MCU) de los que contamos con la información de nombre del personaje y año de la primera película en la que apareció; a partir de estos resolver las siguientes actividades: indicar quien fue el primer y el último personaje en aparecer en una película sin realizar un recorrido de la lista (podrían ser más de uno tanto el primero como el último).

```
#include <iostream>
#include <cstring>
using namespace std;

class Personaje {
private:
    char nombre[50];
    int anio;

public:
    Personaje() {
        nombre[0] = '\0';
        anio = 0;
    }

    void ingresar() {
        cout << "Nombre del personaje: ";
        cin >> nombre;
```

```
        cout << "Año de primera aparición: ";
        cin >> anio;
    }

    int getAnio() { return anio; }
    const char* getNombre() { return nombre; }
};

class MCU {
private:
    Personaje lista[100];
    char primeros[100][50];
    char ultimos[100][50];
    int n, cantPrimeros, cantUltimos;
    int anioMin, anioMax;

public:
    MCU() {
        n = 0;
        cantPrimeros = 0;
        cantUltimos = 0;
        anioMin = 9999;
        anioMax = -1;
    }

    void ingresarPersonajes(int cantidad) {
        n = cantidad;

        for (int i = 0; i < n; i++) {
            cout << "\nPersonaje #" << (i + 1) << "\n";
            lista[i].ingresar();

            int anio = lista[i].getAnio();
            const char* nombre = lista[i].getNombre();

            // Actualizar primeros
            if (anio < anioMin) {
                anioMin = anio;
                cantPrimeros = 0;
                strcpy(primeros[cantPrimeros++], nombre);
            } else if (anio == anioMin) {
                strcpy(primeros[cantPrimeros++], nombre);
            }

            // Actualizar últimos
            if (anio > anioMax) {
                anioMax = anio;
                cantUltimos = 0;
                strcpy(ultimos[cantUltimos++], nombre);
            } else if (anio == anioMax) {
```

```
            strcpy(ultimos[cantUltimos++], nombre);
        }
      }
    }

    void mostrarResultados() {
      cout << "\n===== Primeros personajes del MCU =====\n";
      for (int i = 0; i < cantPrimeros; i++)
        cout << primeros[i] << " (Año " << anioMin << ")\n";

      cout << "\n===== Últimos personajes del MCU =====\n";
      for (int i = 0; i < cantUltimos; i++)
        cout << ultimos[i] << " (Año " << anioMax << ")\n";
    }
};

int main() {
    MCU universo;
    int n;

    cout << "¿Cuántos personajes desea ingresar? ";
    cin >> n;

    universo.ingresarPersonajes(n);
    universo.mostrarResultados();

    return 0;
}
```

8. Se dispone de una lista de películas con la siguiente información: título, año de estreno, recaudación y valoración del público (de 1 a 5), los cuales debemos procesar contemplando las siguiente tareas: determinar si la película “Avengers: Infinity War” está en la lista y mostrar toda su información; indicar en qué posición se encuentra la película “Star Wars: The Return of Jedi”.

```
#include <iostream>
#include <string>
using namespace std;

struct Pelicula {
    string titulo;
    int anio;
    double recaudacion;
    double valoracion;
};

// Función de búsqueda binaria por título
int busquedaBinaria(Pelicula* lista, int n, string objetivo) {
```

```
    int low = 0;
    int high = n - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (lista[mid].titulo == objetivo)
            return mid; // encontrado
        else if (lista[mid].titulo < objetivo)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // no encontrado
}

int main() {
    // Lista ordenada alfabéticamente por título
    Pelicula lista[] = {
        {"Avengers: Infinity War", 2018, 2048.4, 4.8},
        {"Iron Man", 2008, 585.2, 4.5},
        {"Star Wars: The Return of Jedi", 1983, 475.1, 4.7}
    };
    int n = sizeof(lista)/sizeof(lista[0]);

    // Para búsqueda binaria, debemos **ordenar la lista alfabéticamente**
    // (aquí ya está ordenada manualmente para el ejemplo)

    // 1️⃣ Buscar "Avengers: Infinity War"
    int posAvengers = busquedaBinaria(lista, n, "Avengers: Infinity War");
    if (posAvengers != -1) {
        cout << "Avengers: Infinity War encontrada:\n";
        cout << "Título: " << lista[posAvengers].titulo << "\n";
        cout << "Año: " << lista[posAvengers].anio << "\n";
        cout << "Recaudación: " << lista[posAvengers].recaudacion << "\n";
        cout << "Valoración: " << lista[posAvengers].valoracion << "\n";
    } else {
        cout << "Avengers: Infinity War no encontrada.\n";
    }

    // 2️⃣ Buscar "Star Wars: The Return of Jedi"
    int posStarWars = busquedaBinaria(lista, n, "Star Wars: The Return of Jedi");
    if (posStarWars != -1) {
        cout << "\nStar Wars: The Return of Jedi está en la posición: " << posStarWars + 1 << "\n";
    } else {
        cout << "Star Wars: The Return of Jedi no encontrada.\n";
    }

    return 0;
}
```

9. Un vector T tiene cien posiciones, 0.100. Supongamos que las claves de búsqueda de los elementos de la tabla son enteros positivos (por ejemplo, número del DNI).

   Una función de conversión h debe tomar un número arbitrario entero positivo x y convertirlo en un entero en el rango 0.100, esto es, h es una función tal que para un entero positivo x.

   h(x)= n, donde n es entero en el rango 0.100
   El método del módulo, tomando 101, será
   h(x) = x mod 101
   Si se tiene el DNI número 234661234, por ejemplo, se tendrá la posición 56:
   234661234 mod 101 = 56

```
#include <iostream>
using namespace std;

int main() {
   long long dni;
   long long posicion;

   cout << "Ingrese un DNI (entero positivo): ";
   cin >> dni;

   if (dni < 0) {
      cout << "El DNI debe ser un entero positivo." << endl;
      return 0;
   }

   posicion = dni % 101;

   cout << "La posicion hash para el DNI es: " << posicion << endl;

   return 0;
}
```

10. Un entero de ocho dígitos se puede dividir en grupos de tres, tres y dos dígitos, los grupos se suman juntos y se truncan si es necesario para que estén en el rango adecuado de índices.

```
#include <iostream>
using namespace std;

int main() {
   long long dni;
   cout << "Ingrese un DNI de 8 digitos: ";
   cin >> dni;
```

```
    long long* p = &dni;

    long long temp = *p;
    long long* q = &temp;

    int g3 = (int)(*q % 100);
    *q /= 100;

    int g2 = (int)(*q % 1000);
    *q /= 1000;

    int g1 = (int)(*q);

    int suma = g1 + g2 + g3;

    int direccion = suma % 1000;

    cout << "Direccion: " << direccion << endl;

    return 0;
}
```