

**Universidad Nacional del Altiplano**  
**Facultad de Ingeniería Estadística e Informática**  
**Docente:** Fred Torres Cruzz  
**Alumno:** Roberto Angel Ticona Miramira

### Trabajo Encargado - N°009

**Enlace a repositorio de Github:** Repositorio Calculando-tiempo-de-ejecución en GitHub

A continuación se presentaran los códigos utilizados para aplicar la lectura del archivo con datos desordenados, su ordenamiento, aplicación del Jump search, calculo del tiempo aplicado por tipo de búsqueda y determinar que datos demandan menos y mas tiempo al ser ejecutados

1. Muestra de datos desordenados y el archivo txt.

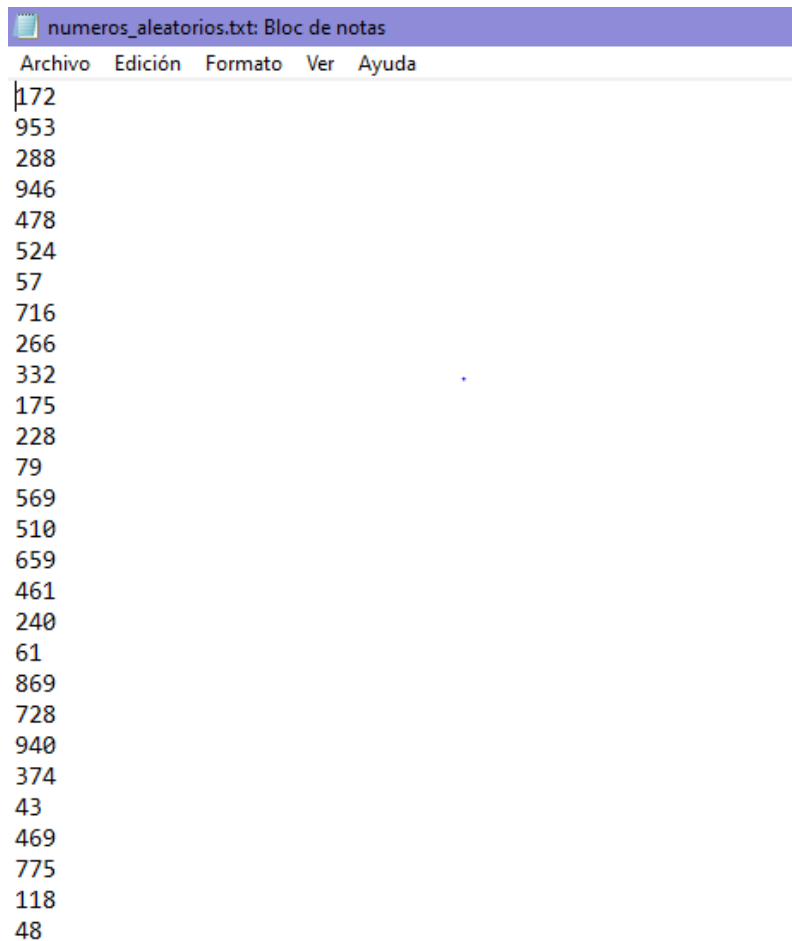


Figura 1: Datos desordenados

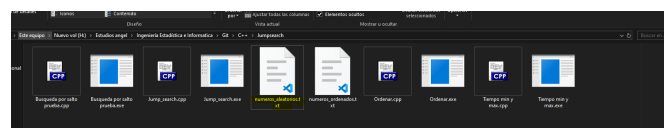
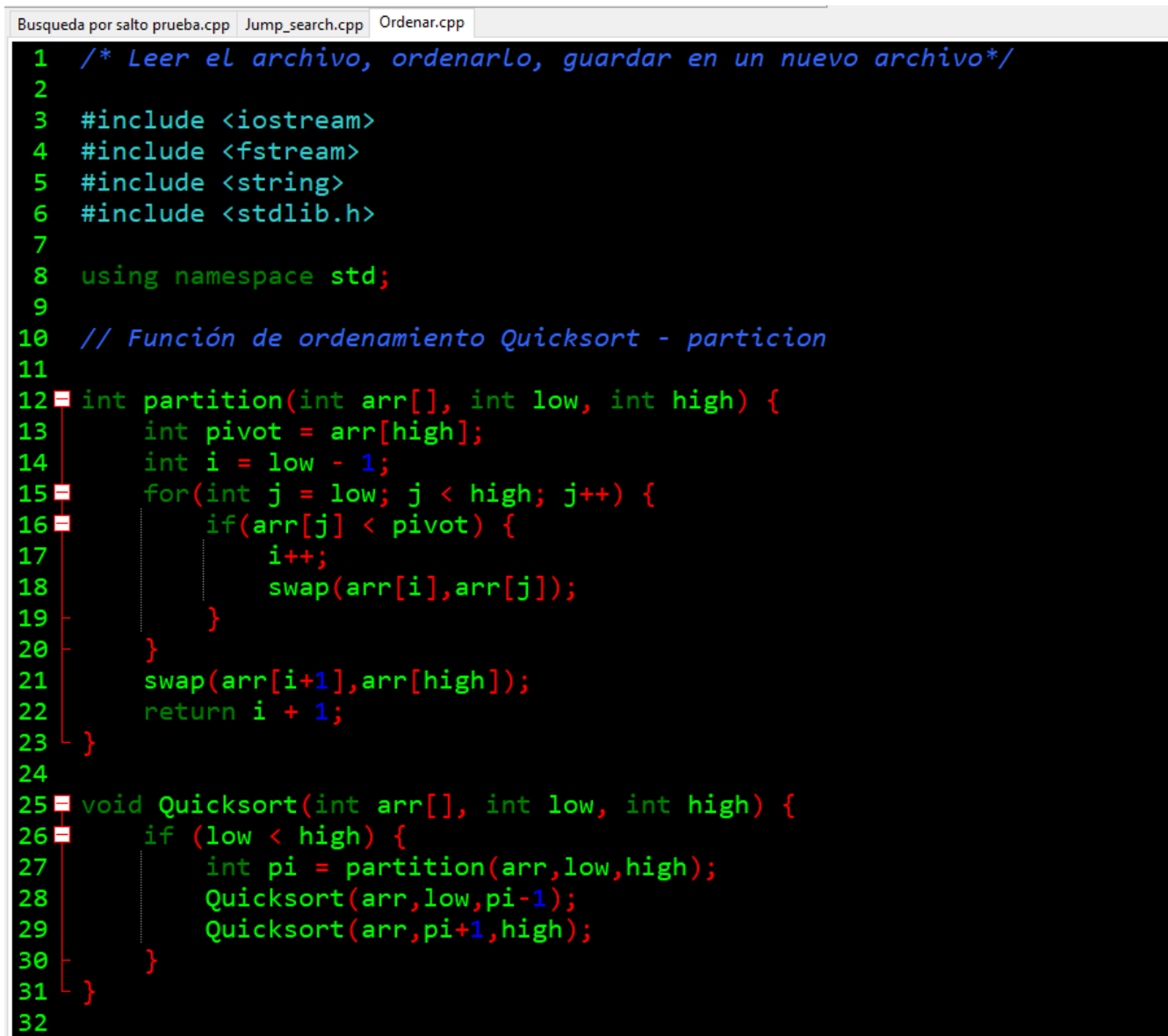


Figura 2: Archivo numeros aleatorios.txt

## 2. Ordenamiento Quicksort aplicado



```
1  /* Leer el archivo, ordenarlo, guardar en un nuevo archivo*/
2
3  #include <iostream>
4  #include <fstream>
5  #include <string>
6  #include <stdlib.h>
7
8  using namespace std;
9
10 // Función de ordenamiento Quicksort - particion
11
12 int partition(int arr[], int low, int high) {
13     int pivot = arr[high];
14     int i = low - 1;
15     for(int j = low; j < high; j++) {
16         if(arr[j] < pivot) {
17             i++;
18             swap(arr[i],arr[j]);
19         }
20     }
21     swap(arr[i+1],arr[high]);
22     return i + 1;
23 }
24
25 void Quicksort(int arr[], int low, int high) {
26     if (low < high) {
27         int pi = partition(arr,low,high);
28         Quicksort(arr,low,pi-1);
29         Quicksort(arr,pi+1,high);
30     }
31 }
32
```

Figura 3: Código en C++ - Función Quicksort

## 3. Lectura del archivo y aplicación del Quicksort

```
33 // Leer archivo
34
35 int main() {
36     ifstream archivo;
37     string frase; // cadena para almacenar los numeros
38     int num[300000];
39     int contador = 0;
40
41     archivo.open("numeros_aleatorios.txt", ios::in);
42     if(archivo.fail()) {
43         cout << "Error al leer el archivo" << endl;
44         return 1;
45     }
46
47     while(getline(archivo, frase)) {
48         num[contador] = stoi(frase);
49         contador++;
50     }
51     archivo.close();
52
53     cout << "Datos guardados: " << contador << " numeros leidos." << endl;
54
55     // Datos ordenados
56
57     Quicksort(num, 0, contador - 1);
58
59     cout << "Numeros ordenados: " << endl;
60     for(int i = 0; i < contador; i++) {
61         cout << num[i] << endl;
62     }
63     cout << endl;
64 }
```

Figura 4: Código en C++ - Lectura del archivo y ordenamiento

## 4. Creación de nuevo archivo con datos ordenados

```
64
65 // Guardar Los números ordenados en un nuevo archivo
66
67 ofstream archivo_salida("numeros_ordenados.txt");
68 if (archivo_salida.fail()) {
69     cout << "Error al crear el archivo de salida" << endl;
70     return 1;
71 }
72
73 for(int i = 0; i < contador; i++) {
74     archivo_salida << num[i] << endl;
75 }
76 archivo_salida.close();
77
78 cout << "Numeros ordenados guardados en 'numeros_ordenados.txt' << endl;
79 system("pause");
80 return 0;
81 }
```

Figura 5: Código en C++ - Crear nuevo archivo

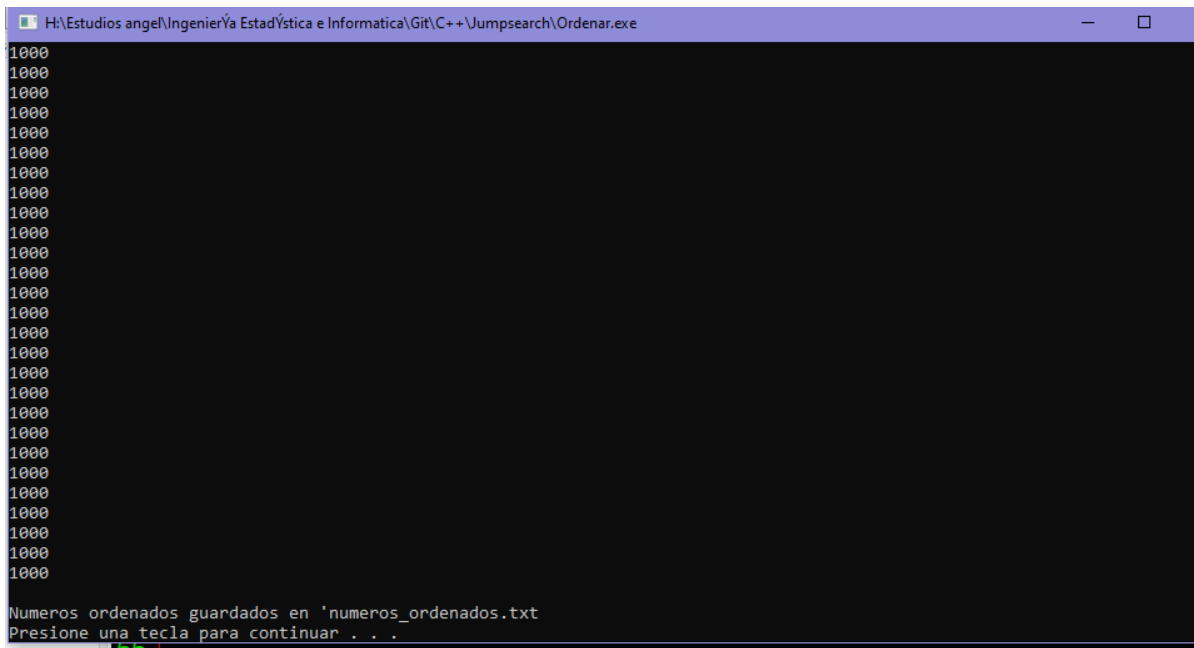


Figura 6: Compilación del código



Figura 7: Datos ordenados

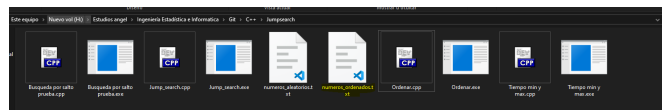


Figura 8: Archivo numeros ordenados.txt

5. Función Jumpsearch para realizar la búsqueda de elementos en el arreglo

```
Jump_search.cpp | Tiempo min y max.cpp
1  #include <bits/stdc++.h>
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  #include <stdlib.h>
6  #include <chrono> // Para medir el tiempo
7  #include <iomanip> // Para configurar precisión en los decimales
8
9  using namespace std;
10
11 // Función de Jump Search
12 int jumpSearch(int arr[], int x, int n) {
13     int paso = sqrt(n);
14     int prev = 0;
15
16     while (arr[min(paso, n) - 1] < x) {
17         prev = paso;
18         paso += sqrt(n);
19         if (prev >= n)
20             return -1;
21     }
22     while (arr[prev] < x) {
23         prev++;
24         if (prev == min(paso, n))
25             return -1;
26     }
27     if (arr[prev] == x)
28         return prev;
29
30     return -1;
31 }
32
```

Figura 9: Función Jumpsearch en C++.txt

6. También se implemento una función para determinar cuantas veces se repetía un dato en el arreglo

```
33 // Función para contar repeticiones
34 int repeticiones(int arr[], int n, int index, int x) {
35     int count = 1;
36
37     // Contar hacia atrás
38     int i = index - 1;
39     while (i >= 0 && arr[i] == x) {
40         count++;
41         i--;
42     }
43
44     // Contar hacia adelante
45     int j = index + 1;
46     while (j < n && arr[j] == x) {
47         count++;
48         j++;
49     }
50
51     return count;
52 }
53
```

Figura 10: Función repeticiones en C++.txt

## 7. Lectura del nuevo archivo con datos ordenados y aplicación del algoritmo de ordenamiento por Jumpsearch

```

60 // Abrir archivo
61 archivo.open("numeros_ordenados.txt", ios::in);
62 if (archivo.fail()) {
63     cout << "Error al leer el archivo" << endl;
64     return 1;
65 }
66
67 // Leer archivo y guardar los números
68 while (getline(archivo, frase)) {
69     num[contador] = stoi(frase);
70     contador++;
71 }
72 archivo.close();
73 cout << "Datos guardados: " << contador << " numeros leidos" << endl;
74
75 // Solicitar número a buscar
76 int x;
77 cout << "Digite el numero a buscar: ";
78 cin >> x;
79
80 // Repetir el algoritmo para amplificar el tiempo de ejecución
81 const int REPETICIONES = 100000; // Número de repeticiones
82 auto start_time = chrono::high_resolution_clock::now();
83

```

Figura 11: Abrir nuevo archivo y búsqueda en C++.txt

## 8. Como se tenia que calcular el tiempo de ejecución se agrego la constante para que realice 100000 repeticiones y determinar cuanto tiempo se demora y calcular el tiempo de ejecución total

```

80 // Repetir el algoritmo para amplificar el tiempo de ejecución
81 const int REPETICIONES = 100000; // Número de repeticiones
82 auto start_time = chrono::high_resolution_clock::now();
83
84 int indice = -1;
85 for (int i = 0; i < REPETICIONES; ++i) {
86     indice = jumpSearch(num, x, contador);
87 }
88
89 auto end_time = chrono::high_resolution_clock::now();
90
91 // Calcular tiempo de ejecución total y promedio
92 chrono::duration<double> total_time = end_time - start_time;
93 double tiempo_promedio = total_time.count() / REPETICIONES;
94
95 int ocurrencias = 0;
96 if (indice != -1) {
97     ocurrencias = repeticiones(num, contador, indice, x);
98 }
99

```

Figura 12: Determinar el tiempo de 100000 repeticiones de la función y el tiempo normal de búsqueda

## 9. Mostrar resultados y compilación

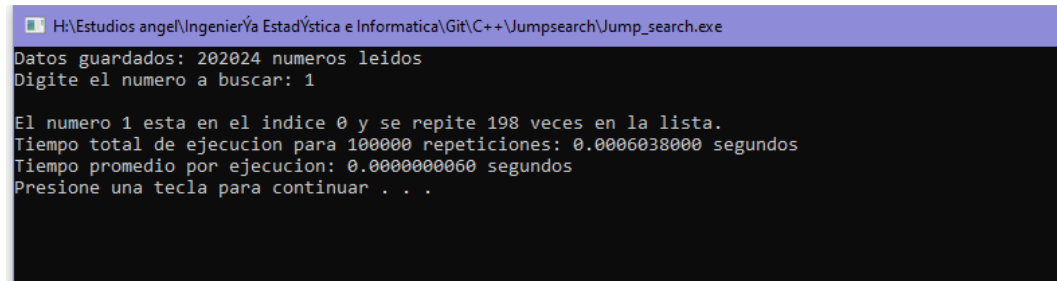
```

100 // Mostrar resultados
101 if (indice != -1) {
102     cout << "\nEl numero " << x << " esta en el indice " << indice;
103     cout << " y se repite " << ocurrencias << " veces en la lista." << endl;
104 } else {
105     cout << "\nEl numero " << x << " no se encuentra en la lista." << endl;
106 }
107
108 // Mostrar tiempo total y promedio
109 cout << fixed << setprecision(10); // Mostrar 10 decimales
110 cout << "Tiempo total de ejecucion para " << REPETICIONES << " repeticiones: "
111     << total_time.count() << " segundos" << endl;
112 cout << "Tiempo promedio por ejecucion: " << tiempo_promedio << " segundos" << endl;
113
114 system("pause");
115 return 0;
116 }
117

```

Figura 13: Mostrar resultados obtenidos

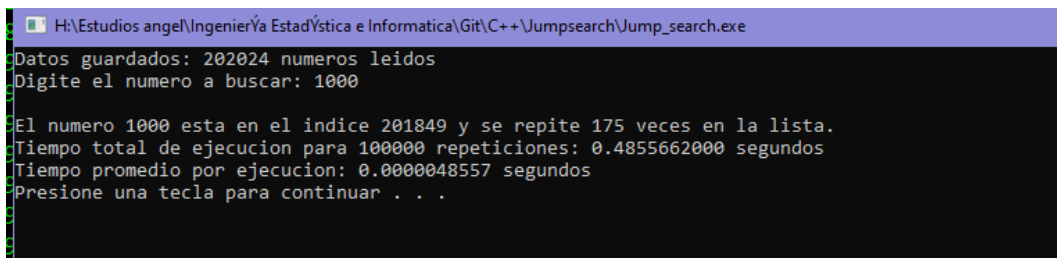
## 10. Resultados de pruebas



```
H:\Estudios angel\Ingeniería Estadística e Informática\Git\C++\Jumpsearch\Jump_search.exe
Datos guardados: 202024 numeros leídos
Digite el numero a buscar: 1

El numero 1 esta en el indice 0 y se repite 198 veces en la lista.
Tiempo total de ejecucion para 100000 repeticiones: 0.0006038000 segundos
Tiempo promedio por ejecucion: 0.0000000060 segundos
Presione una tecla para continuar . . .
```

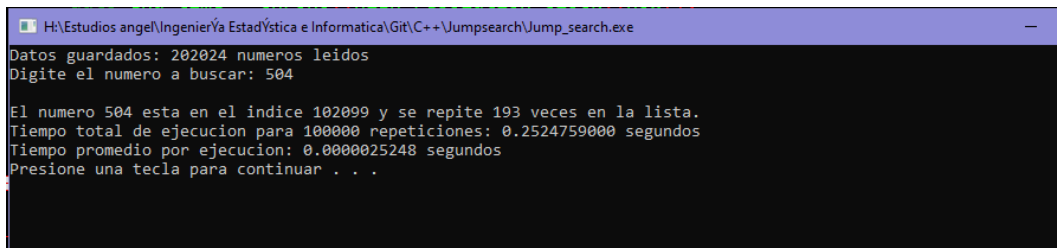
Figura 14: Prueba 1



```
H:\Estudios angel\Ingeniería Estadística e Informática\Git\C++\Jumpsearch\Jump_search.exe
Datos guardados: 202024 numeros leídos
Digite el numero a buscar: 1000

El numero 1000 esta en el indice 201849 y se repite 175 veces en la lista.
Tiempo total de ejecucion para 100000 repeticiones: 0.4855662000 segundos
Tiempo promedio por ejecucion: 0.0000048557 segundos
Presione una tecla para continuar . . .
```

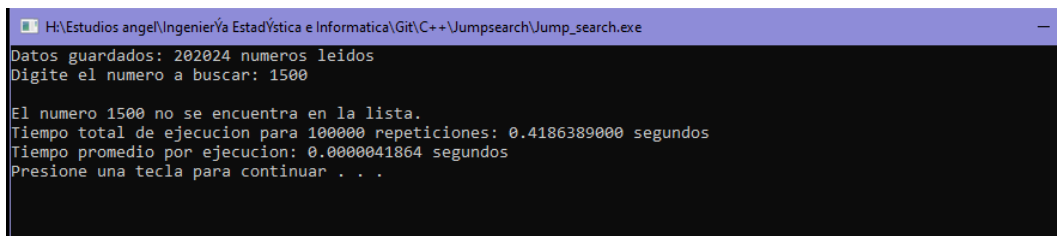
Figura 15: Prueba 2



```
H:\Estudios angel\Ingeniería Estadística e Informática\Git\C++\Jumpsearch\Jump_search.exe
Datos guardados: 202024 numeros leídos
Digite el numero a buscar: 504

El numero 504 esta en el indice 102099 y se repite 193 veces en la lista.
Tiempo total de ejecucion para 100000 repeticiones: 0.2524759000 segundos
Tiempo promedio por ejecucion: 0.0000025248 segundos
Presione una tecla para continuar . . .
```

Figura 16: Prueba 3



```
H:\Estudios angel\Ingeniería Estadística e Informática\Git\C++\Jumpsearch\Jump_search.exe
Datos guardados: 202024 numeros leídos
Digite el numero a buscar: 1500

El numero 1500 no se encuentra en la lista.
Tiempo total de ejecucion para 100000 repeticiones: 0.4186389000 segundos
Tiempo promedio por ejecucion: 0.0000041864 segundos
Presione una tecla para continuar . . .
```

Figura 17: Prueba 4

## 11. Determinación del dato con menor y mayor tiempo de ejecución para la búsqueda en el arreglo

```
Tiempo min y max.cpp
1 #include <bits/stdc++.h>
2 #include <iostream>
3 #include <fstream>
4 #include <string>
5 #include <stdlib.h>
6 #include <chrono> // Para medir el tiempo
7 #include <iomanip> // Para configurar precisión en los decimales
8
9 using namespace std;
10
11 // Función de Jump Search
12 int jumpSearch(int arr[], int x, int n) {
13     int paso = sqrt(n);
14     int prev = 0;
15
16     while (arr[min(paso, n) - 1] < x) {
17         prev = paso;
18         paso += sqrt(n);
19         if (prev >= n)
20             return -1;
21     }
22     while (arr[prev] < x) {
23         prev++;
24         if (prev == min(paso, n))
25             return -1;
26     }
27     if (arr[prev] == x)
28         return prev;
29
30     return -1;
31 }
32
```

Figura 18: Nuevo código para determinar min y max tiempo de ejecución

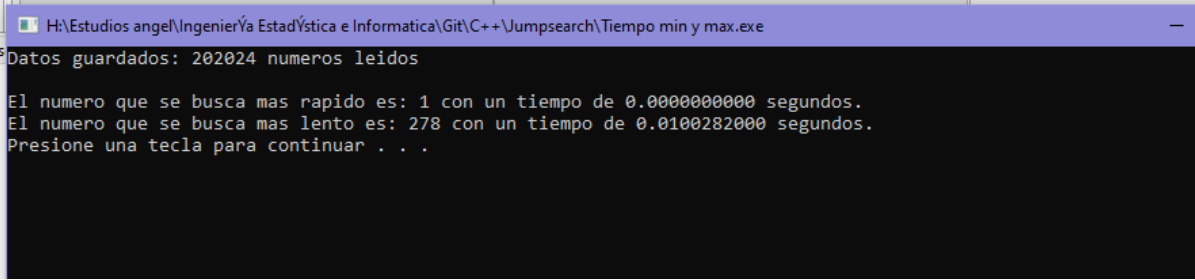
```
32
33 int main() {
34     ifstream archivo;
35     string frase;
36     int num[300000];
37     int contador = 0;
38
39     // Abrir archivo
40     archivo.open("numeros_ordenados.txt", ios::in);
41     if (archivo.fail()) {
42         cout << "Error al leer el archivo" << endl;
43         return 1;
44     }
45
46     // Leer archivo y guardar los números
47     while (getline(archivo, frase)) {
48         num[contador] = stoi(frase);
49         contador++;
50     }
51     archivo.close();
52     cout << "Datos guardados: " << contador << " numeros leidos" << endl;
53 }
```

Figura 19: Leer los datos y guardarlos en un arreglo



```
80
81 // Mostrar resultados
82 cout << fixed << setprecision(10);
83 cout << "\nEl numero que se busca mas rapido es: " << num_rapido
84 | << " con un tiempo de " << tiempo_min << " segundos." << endl;
85 cout << "El numero que se busca mas lento es: " << num_lento
86 | << " con un tiempo de " << tiempo_max << " segundos." << endl;
87
88 system("pause");
89 return 0;
90 }
91
```

Figura 20: Mostrar resultados



```
H:\Estudios angel\Ingeniería Estadística e Informática\Git\C++\Jumpsearch\Tiempo min y max.exe
Datos guardados: 202024 numeros leidos
El numero que se busca mas rapido es: 1 con un tiempo de 0.0000000000 segundos.
El numero que se busca mas lento es: 278 con un tiempo de 0.0100282000 segundos.
Presione una tecla para continuar . . .
```

Figura 21: Compilación final