



# Tecnológico de Monterrey

## **Algoritmos de Búsqueda y Ordenamiento**

Ramón Alberto Gómez Urquidez A01254784

10 de septiembre del 2023

Programación de estructuras de datos y algoritmos fundamentales

Profesor Baldomero Olvera Villanueva

# Documentación

## Casos de Prueba

Los casos de prueba se encuentran almacenados independientemente en los archivos *in1.txt*, *in2.txt*, *in3.txt* e *in4.txt*. Al correr el programa, el usuario solo requiere descomentar la línea que desea testear y comentar el resto. Una función denominada *parseIntoVec* se encargará de automatizar el proceso de *parsear* los datos del texto a un vector anidado de enteros.

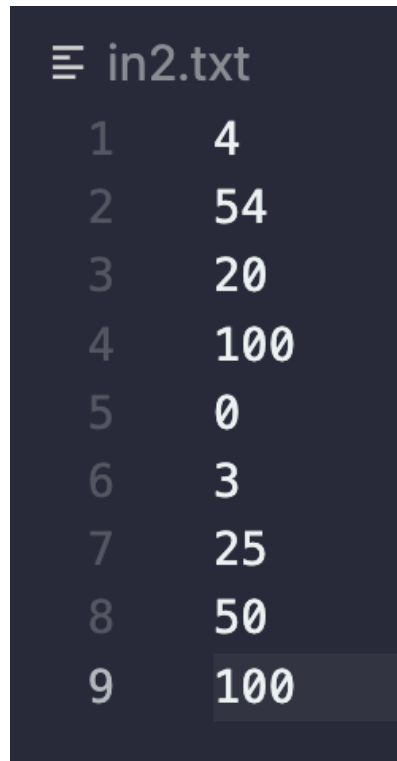
Estos son los siguientes inputs y outputs de los casos de prueba

Input	Output
8 10 4 8 12 20 15 54 18 4 20 54 100 12	28 18 14 6 7 3 7 8 4 -1 8 4 3 4 1
4 54 20 100 0 3 25 50 100	6 6 5 -1 4 2 -1 4 2 3 4 3
10 10 9 8 7 6 5	45 45 15 9 10 4

4 3 2 1 1 10	
0	Error - Vector size must be a positive integer.

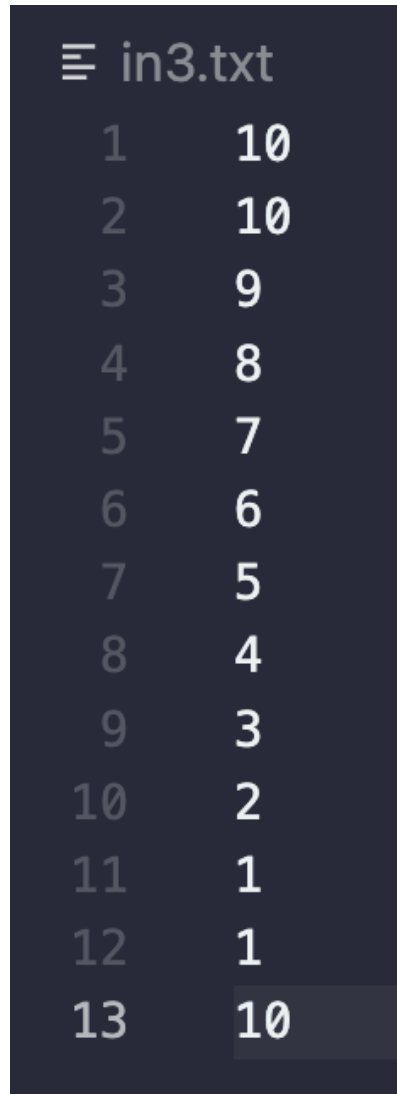
```
≡ in1.txt
1      8
2     10
3      4
4      8
5     12
6     20
7     15
8     54
9     18
10     4
11    20
12    54
13   100
14    12
```

**Figura 1.** Datos almacenados en *in1.txt*



≡ in2.txt	
1	4
2	54
3	20
4	100
5	0
6	3
7	25
8	50
9	100

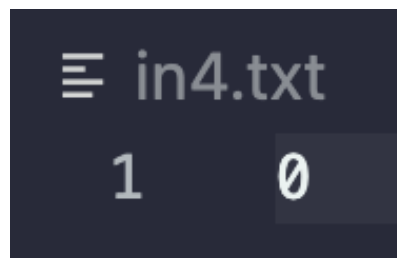
**Figura 2.** Datos almacenados en *in2.txt*



≡ in3.txt

1	10
2	10
3	9
4	8
5	7
6	6
7	5
8	4
9	3
10	2
11	1
12	1
13	10

**Figura 3.** Datos almacenados en *in3.txt*



≡ in4.txt

1	0
---	---

**Figura 4.** Datos almacenados en *in4.txt*

```

parse > parseIntoVec.cpp > parseIntoVec(std::string)
1  #include "parseIntoVec.hpp"
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  #include <vector>
6  #include <stdexcept>
7
8  std::vector<std::vector<int>> parseIntoVec(std::string fileName) {
9      std::vector<std::vector<int>> parsedVec;
10     std::ifstream inputFile;
11     int sizeVec, num;
12     inputFile.open(fileName); //RAII is responsible for destruct the file-handling object out of scope. It will handle file closing automatically
13
14     if (!inputFile.is_open()) {
15         throw std::runtime_error("Error - Unable to open the file.");
16     }
17
18     while (inputFile >> sizeVec) {
19         if (sizeVec <= 0) {
20             throw std::runtime_error("Error - Vector size must be a positive integer.");
21         }
22
23         std::vector<int> tempVec;
24         tempVec.reserve(sizeVec); // To reduced reallocation vector size
25
26         for (int i = 0; i < sizeVec; ++i) {
27             inputFile >> num;
28             tempVec.push_back(num);
29         }
30
31         parsedVec.push_back(tempVec);
32     }
33
34     return parsedVec;
35 }

```

**Figura 5.** Código de la función *parseIntoVec.cpp*

## Funciones

### ordenaIntercambio

El código posee múltiples líneas de código que solo realizan una operación (como la inicialización del count, el incremento de este o las operaciones adentro de la declaración comparativa). Lo interesante de este código es que posee dos for anidados que son empleados para calcular el intercambio de datos. Con esto en mente, podemos decir que:

$$2O(1) + O(n) + 6O(n^2) = 2 + n + 6n^2 = 6n^2 = n^2$$

donde

- $O(1)$  = En la declaración del contador y en el return
- $O(n)$  = En el primer loop
- $O(n^2)$  = En el segundo loop + las cinco operaciones adentro de este

```
sort > C++ exchangeSort.cpp > ordenaIntercambio(std::vector<int>&)  
1  ∨ #include "exchangeSort.hpp"  
2    #include <iostream>  
3  
4  ∨ int ordenaIntercambio(std::vector<int>& vec) {  
5      int count{};  
6  
7      for (int i = 0; i < vec.size() - 1; ++i) {  
8          for (int j = i+1; j < vec.size(); ++j) {  
9              ++count;  
10             if (vec[i] > vec[j]) {  
11                 int aux = vec[i];  
12                 vec[i] = vec[j];  
13                 vec[j] = aux;  
14             }  
15         }  
16     }  
17  
18     return count;  
19 }
```

Figura 6. Código de la función *ordenaIntercambio()*

## ordenaBurbuja

El código posee múltiples líneas de código que solo realizan una sola operación, posee un ciclo *do-while* y un ciclo for loop que hará que el código tenga que iterarse de forma anidada. Esto se debe a que en cada iteración del bucle exterior, se realizan  $n - 1$  comparaciones en el bucle interior. Con esto en mente, podemos decir que:

$$4O(1) + 3O(n) + 7O(n^2) = 4 + 3n + 7n^2 = 7n^2 = n^2$$

donde

- $O(1)$  = En la declaración del contador, de la *k*, del *switchSwap* y del *return*
- $O(n)$  = En el primer *do-while*, cuando el *switchSwap* pasa a ser falso y el incremento
- $O(n^2)$  = En el for loop + las seis operaciones adentro de este

```
sort > C++ bubbleSort.cpp > ordenaBurbuja(std::vector<int>&)  
1  #include "bubbleSort.hpp"  
2  
3  int ordenaBurbuja(std::vector<int>& vec) {  
4      int count{}, k = 1;  
5      bool switchSwap;  
6  
7      do {  
8          switchSwap = false;  
9          for (int i = 0; i < vec.size() - k; ++i) {  
10             if (vec[i] > vec[i+1]) {  
11                 int aux = vec[i];  
12                 vec[i] = vec[i+1];  
13                 vec[i+1] = aux;  
14                 switchSwap = true;  
15             }  
16             ++count;  
17         }  
18         ++k;  
19     } while (switchSwap);  
20  
21     return count;  
22 }
```

Figura 7. Código de la función *ordenaBurbuja()*

## ordenaMerge

En base al desglose matemático (teorema maestro) para calcular merge sort, damos con la siguiente información simplificada

$$T(n) = 2^k T(n/2^k) + kcn$$

$$T(n/2) = 2^k T(n/4^k) + kcn$$

$$T(n/4) = 2^k T(n/8^k) + kcn$$

$$T(n) = o(n \log n)$$

donde

- $T(n)$  = Es la función que representa la complejidad de tiempo de algún algoritmo
- $n/2^k$  = Representa el tamaño del problema por cada división. Se asume que en la recurrencia el problema es dividido en subproblemas de esa longitud, y  $k$  es un parámetro que indica la cantidad de veces que se divide la lista original
- $kcn$  = El tiempo necesario para combinar las sublistas ordenadas



```
sort > G+ mergeSort.cpp > ...
1  #include "mergeSort.hpp"
2  #include <iostream>
3
4  int merge(std::vector<int>& vec, int left, int mid, int right, int& count) {
5      int n1 = mid - left + 1, n2 = right - mid;
6      std::vector<int> leftvec(n1), rightvec(n2);
7
8      for (int i = 0; i < n1; i++) {
9          leftvec[i] = vec[left + i];
10     }
11     for (int i = 0; i < n2; i++) {
12         rightvec[i] = vec[mid + 1 + i];
13     }
14
15     int i = 0, j = 0, k = left;
16
17     while (i < n1 && j < n2) {
18         count++;
19         if (leftvec[i] <= rightvec[j]) {
20             vec[k] = leftvec[i];
21             i++;
22         } else {
23             vec[k] = rightvec[j];
24             j++;
25         }
26         k++;
27     }
28
29     while (i < n1) {
30         vec[k] = leftvec[i];
31         i++; k++;
32     }
33
34     while (j < n2) {
35         vec[k] = rightvec[j];
36         j++; k++;
37     }
38
39     return count;
40 }
```

**Figura 8.** Código de la función *merge()*

```
42  int ordenaMerge(std::vector<int>& vec, int left, int right, int& count) {
43      if (left < right) {
44          int mid = left + (right - left) / 2;
45          ordenaMerge(vec, left, mid, count);
46          ordenaMerge(vec, mid + 1, right, count);
47          return merge(vec, left, mid, right, count);
48      }
49      return 0;
50  }
```

Figura 9. Código de la función *ordenaMerge()*

## busqSecuencial

El código posee una línea para inicializar el contador y dos returns, así como un for-loop que itera por todos los elementos del contenedor (vector). Con esto en mente, podemos decir que:

$$3O(1) + 3O(n) = 3 + 3n = 3n = n$$

donde

- $O(1)$  = En la declaración del contador y los dos returns (aunque este anidado uno de los return en el for loop, solo se ejecutará una y solo una vez en el código por la declaración)
- $O(n)$  = En el for loop + las dos operaciones adentro de este

```
search > G+ sequentialSearch.cpp > busqSecuencial(const std::vector<int>&, int)
1  #include "sequentialSearch.hpp"
2  #include <vector>
3
4  int busqSecuencial(const std::vector<int>& vec, int target) {
5      int count{};
6      for (auto &i : vec) {
7          ++count;
8          if (i == target) {
9              return count;
10         }
11     }
12     return count;
13 }
```

Figura 10. Código de la función *busqSecuencial()*

## busqBinaria

El código se caracteriza por dividir la mitad del vector constantemente hasta hallar el índice del objeto en cuestión. Con esta pequeña lógica, y dejando a un lado las otras operaciones, se puede simplificar el cálculo de la siguiente manera (como si fuera recursión para mejor entendimiento)

$$\begin{aligned}T(n) &= c + T(n/2) \\T(n/2) &= c + T(n/4) \\T(n/4) &= c + T(n/8) \\T(n) &= c + T(n/2^i) + ic \\T(n/2^i) &= T(1) \\n/2^i &= 1 \\n &= 2^i \\log_2 n &= \log_2 2^i \\log_2 n &= i\end{aligned}$$

$$T(n) = T(n/2^{\log_2 n}) + c \log_2 n \text{ (el denominador es igual a } n \text{)}$$

$$T(n) = T(n/n) + c \log_2 n$$

$$T(n) = T(1) + c \log_2 n$$

$$T(n) = o(\log_2 n)$$

donde

- $T(n)$  = La función.
- $C$  = La constante del tiempo, que representa las operaciones basicas en cada nivel de repetición o recursión.
- $T(n/2)$  = La función con la mitad de datos.

```
search > G+ binarySearch.cpp > busqBinaria(const std::vector<int>&, int)
1  #include "binarySearch.hpp"
2  #include <vector>
3  #include <iostream>
4
5  Values busqBinaria(const std::vector<int>& vec, int target) {
6      int left = 0, right = vec.size() - 1, count{};
7
8      while (left <= right) {
9          ++count;
10         int mid = left + (right - left) / 2;
11         if (vec[mid] == target) {
12             return {mid, count};
13         } else if (vec[mid] < target) {
14             left = mid + 1;
15         } else {
16             right = mid - 1;
17         }
18     }
19
20     return {-1, count};
21 }
```

Figura 11. Código de la función *busqBinaria()*

# Main

```
1 #include "../parse/parseIntoVec.hpp"
2 #include "../sort/exchangeSort.hpp"
3 #include "../sort/bubbleSort.hpp"
4 #include "../sort/mergeSort.hpp"
5 #include "../search/sequentialSearch.hpp"
6 #include "../search/binarySearch.hpp"
7 #include <iostream>
8 #include <string>
9 #include <vector>
10
11 int main() {
12     int countMerge = 0;
13
14     try {
15         std::vector<std::vector<int>> vecMain = parseIntoVec("in1.txt");
16         /* std::vector<std::vector<int>> vecMain = parseIntoVec("in2.txt"); */
17         /* std::vector<std::vector<int>> vecMain = parseIntoVec("in3.txt"); */
18         /* std::vector<std::vector<int>> vecMain = parseIntoVec("in4.txt"); */
19
20         std::vector<int> sortVector = vecMain[0];
21         std::vector<int> exchangeVector = sortVector;
22         std::vector<int> bubbleVector = sortVector;
23
24         std::cout << ordenaIntercambio(exchangeVector) << " " << ordenaBurbuja(bubbleVector) << " " << ordenaMerge(vecMain[0], 0, vecMain[0].size() - 1, countMerge) << "\n";
25
26         for (auto &&i : vecMain[1]) {
27             auto [index, binCount] = busqBinaria(vecMain[0], i);
28             std::cout << index << " " << busqSecuencial(vecMain[0], i) << " " << binCount << "\n";
29         }
30
31     } catch (const std::exception& e) {
32         std::cerr << e.what() << std::endl;
33         return 1;
34     }
35
36     return 0;
37 }
```

Figura 12. Código del *main.cpp*

## Output

```
M makefile
1  CXX = g++
2  CXXFLAGS = -Wall -std=c++17
3  TARGET = main
4  PRSDIR = parse
5  SRCDIR = search
6  SRTDIR = sort
7
8  SRCS = main.cpp \
9         $(PRSDIR)/parseIntoVec.cpp \
10        $(SRCDIR)/sequentialSearch.cpp \
11        $(SRCDIR)/binarySearch.cpp \
12        $(SRTDIR)/exchangeSort.cpp \
13        $(SRTDIR)/bubbleSort.cpp \
14        $(SRTDIR)/mergeSort.cpp
15
16  EXECUTABLE = a.out
17
18  all: $(EXECUTABLE)
19
20  $(EXECUTABLE): $(SRCS)
21      $(CXX) $(CXXFLAGS) $^ -o $@
22
23  run: $(EXECUTABLE)
24      ./$(EXECUTABLE)
25
26  clean:
27      rm -f $(EXECUTABLE)
28
29  .PHONY: all run clean
```

**Figura 13.** Archivo makefile para comandos de compilación en G++. Solo requiere de usar el comando *make clean*, *make all* y *make run* cada vez que quiera compilar y ejecutar un programa.

```
> make run
./a.out
28 18 14
6 7 3
7 8 4
-1 8 4
3 4 1
```

**Figura 14.** Output del primer caso de prueba.

```
> make run
./a.out
6 6 5
-1 4 2
-1 4 2
3 4 3
```

**Figura 15.** Output del segundo caso de prueba.

```
> make run  
./a.out  
45 45 15  
9 10 4
```

**Figura 16.** Output del tercer caso de prueba.

```
> make run  
./a.out  
Error – Vector size must be a positive integer.  
make: *** [run] Error 1
```

**Figura 17.** Output del cuarto caso de prueba.