

# Sort and Search Algorithms

Santiago Toll Leyva  
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: [idv16a.stoll@uartesdigitales.edu.mx](mailto:idv16a.stoll@uartesdigitales.edu.mx)

Profesor: Efraín Padilla

# Tarea 2: Algoritmos de ordenamiento

## Teoria

Los algoritmos de ordenamiento tienen la función de acomodar una serie de elementos en un orden dado. Estos algoritmos necesitan una entrada; la función de estos algoritmos es regresar una salida que contenga los elementos de la entrada organizados en base al orden especificado.

Los algoritmos de búsqueda se encargan de encontrar un elemento dentro de una serie de elementos. Distintos tipos de algoritmos son mejores dependiendo del orden de la serie. Algunos de estos algoritmos utilizan recursividad mientras que otros evalúan cada elemento en orden. Pero ambos tipos son útiles dependiendo de las necesidades.

## Planteamiento del problema

Ordenar una serie de elementos puede llegar a ser un proceso tardado. Dependiendo de la magnitud de la entrada y la cantidad de elementos que deberán ser ordenados, estos algoritmos aumentan su costo.

Encontrar un elemento específico en una serie puede llegar a ser tardado, pero esto depende de cómo se encuentra ordenada la serie y cuántos elementos contiene.

## Solución del problema

Ya que estos algoritmos siempre tendrán un costo, se decidió evaluar cinco de ellos utilizando tres tipos distintos de entrada. Cada entrada es un vector que contiene 500 elementos. El primer vector contiene todos los elementos ordenados de manera ascendente; el segundo vector contiene todos los elementos ordenados de manera descendente; el tercer vector contiene elementos generados de manera aleatoria.

De mismo modo se evaluarán las funciones de búsqueda (binaria y lineal) para poder observar el comportamiento de ambos algoritmos.

En el caso de los algoritmos de ordenamiento:

Para evaluar el tiempo que tarda cada algoritmo en ordenar estos vectores, se iteró diez veces por cada elemento para poder obtener un promedio del tiempo en ejecución. Este proceso se repite varias veces utilizando vectores de distintos tamaños (vacío hasta 500).

Se hará lo mismo con los algoritmos de búsqueda, pero se buscará el mismo elemento una y otra vez para obtener el promedio que se tarda en encontrar ese elemento.

## Codigo

Funciones utilizadas para generar vectores.

Listing 1. Vector Generation Functions C++

---

```

1  /* ***** */
2  /* Vector Generator functions */
3  /* ***** */
4  vector<int> ascendingVector(int size) {
5      vector<int> returnVector;
6      for (int i = 0; i < size; i++) {
7          returnVector.push_back(i);
8      }
9      return returnVector;
10 }
11
12 vector<int> descendingVector(int size) {
13     vector<int> returnVector;
14     for (int i = size; i > 0; i--) {
15         returnVector.push_back(i);
16     }
17     return returnVector;
18 }
19
20 vector<int> randomVector(int size, int min, int max) {
21     vector<int> returnVector;
22     std::srand(std::time(0));
23     for (int i = 0; i < size; i++) {
24         int random = min + rand() % ((max + 1) - min); //Generate a random numbver between min and
25
26         returnVector.push_back(random);
27     }
28     return returnVector;
29 }

```

---

Listing 2. Quick Sort Algorithm C++

---

```

1  /* **** */
2  /* Quick Sort functions */
3  /* **** */
4  int partition(vector<int>& vector, int low, int high) {
5      int pivot = vector[high]; // Find the pivot for this iteration.
6      int i = low - 1;
7      for (int j = low; j < high; j++) {
8          if (vector[j] < pivot) {
9              i++;
10             std::swap(vector[i], vector[j]); // Swap the positions
11         }
12     }
13     std::swap(vector[i + 1], vector[high]); // Swap the positions
14     return(i + 1);
15 }
16
17 void quickSort(vector<int>& vector, int low, int high) {
18     if (vector.size() < 2) {
19         return;
20     }
21     if (low < high) {
22         int partitionIndex = partition(vector, low, high);
23         // Recursivity to travel the binary tree
24         quickSort(vector, low, partitionIndex - 1);
25         quickSort(vector, partitionIndex + 1, high);
26     }
27 }
28 void quickSortCall(vector<int>& vector) {
29     quickSort(vector, 0, vector.size() - 1);
30 }

```

---

## Algoritmo Merge sort.

Listing 3. Merge Sort Algorithm C++

---

```

1  /* **** */
2  /* Merge Sort functions */
3  /* **** */
4  void merge(vector<int>& inputVector , int left , int middle , int right) {
5      int leftIndex; //Index of the first subarray.
6      int rightIndex; //Index of the second subarray.
7      int mergeIndex; //Index of the merged subarray.
8
9      int leftSize = middle - left + 1;
10     int rightSize = right - middle;
11
12     vector<int> leftVector , rightVector; //Temporary vectors.
13
14     for (leftIndex = 0; leftIndex < leftSize; leftIndex++) {
15         leftVector.push_back(inputVector.at(left + leftIndex));
16     }
17     for (rightIndex = 0; rightIndex < rightSize; rightIndex++) {
18         rightVector.push_back(inputVector.at(middle + 1 + rightIndex));
19     }
20     //Set the indices.
21     leftIndex = 0;
22     rightIndex = 0;
23     mergeIndex = left;
24
25     //Merge the temporary vectors into original.
26     while (leftIndex < leftSize && rightIndex < rightSize)
27     {
28         if (leftVector[leftIndex] <= rightVector[rightIndex]) {
29             inputVector[mergeIndex] = leftVector[leftIndex];
30             leftIndex++;
31         }
32         else {
33             inputVector[mergeIndex] = rightVector[rightIndex];
34             rightIndex++;
35         }
36     }
37     //Copy remaining elements(if any)
38     while (leftIndex < leftSize)
39     {
40         inputVector[mergeIndex] = leftVector[leftIndex];
41         leftIndex++;
42         mergeIndex++;
43     }
44     while (rightIndex < rightSize)
45     {
46         inputVector[mergeIndex] = rightVector[rightIndex];
47         rightIndex++;
48         mergeIndex++;
49     }
50 }
51 void mergeSort(vector<int>& vector , int left , int right) {
52     if (vector.size() < 2) {
53         return;
54     }
55     if (left < right) {

```

```
56     int middle = left + (right - left) / 2;
57     // Recursivity
58     mergeSort(vector, left, middle);
59     mergeSort(vector, middle + 1, right);
60
61     merge(vector, left, middle, right);
62 }
63 }
64 void mergeSortCall(vector<int>& vector) {
65     mergeSort(vector, 0, vector.size() - 1);
66 }
```

---

## Binary Search.

Listing 4. Binary Search C++

---

```
1  int binarySearch(vector<int>& vector, int left, int right, int key) {
2      if (vector.size() < 2)
3      {
4          return -900;
5      }
6      if (right < left) {
7          return -999;
8      }
9
10     int mid = left + ((right-left) / 2);
11     if (vector[mid] == key) {
12         return mid;
13     }
14     if (vector[mid] > key) {
15         return binarySearch(vector, left, mid - 1, key);
16     }
17     else if (vector[mid] < key) {
18         return binarySearch(vector, mid + 1, right, key);
19     }
20     else {
21         return mid;
22     }
23 }
```

---

## Linear Search.

Listing 5. Linear Search C++

---

```

1  /* *****/
2  /* Linear search functions */
3  /* *****/
4  int linearSearch(vector<int>& vector , int key) {
5      if (vector.size() < 2) {
6          return -900;
7      }
8      int size = vector.size();
9      for (int i = 0; i<size; i++)
10     {
11         if (vector[i] == key) {
12             return i;
13         }
14     }
15 }
16 vector<int> linearSearchAll(vector<int>& inputVector , int key) {
17     if (inputVector.size() < 2) {
18         return ascendingVector(1);
19     }
20     int size = inputVector.size();
21     vector<int> tempVector;
22     for (int i = 0; i < size; i++)
23     {
24         if (inputVector[i] == key) {
25             tempVector.push_back(i);
26         }
27     }
28     return tempVector;
29 }

```

---



## Algoritmo de Benchmarking.

Listing 6. Benchmarking Algorithm C++

---

```

1  /* ***** */
2  /* Benchmarking functions */
3  /* ***** */
4  template <typename ... Args>
5  void benchmark(int testSize , int iterations , std::function<void(vector<int>&>> func , string fileName) ) {
6      //Create vectors that will be used for benchmarking.
7      const vector<int> bestVector = ascendingVector(testSize);
8      const vector<int> worstVector = descendingVector(testSize);
9      const vector<int> averageVector = randomVector(testSize , 0 , 9);
10     vector<int> usedVector;
11
12     //Create duration variables for each case.
13     duration<float , std::micro> duration;
14
15     float bestDuration = 0;
16     float worstDuration = 0;
17     float averageDuration = 0;
18
19     //Create start and end time so it doesn't happen on every loop.
20     auto startTime = high_resolution_clock::now();
21     auto endTime = high_resolution_clock::now();
22
23     //Initialize file stream.
24     std::ofstream file;
25     string fileText;
26
27     //Write function name at file start.
28     fileText += fileName;
29     fileText += "\n";
30     fileText += "Elements";
31     fileText += ", ";
32     fileText += "Best";
33     fileText += ", ";
34     fileText += "Worst";
35     fileText += ", ";
36     fileText += "Average";
37     fileText += "\n";
38
39     //Iterate case for every input size up to test size.
40     for (int element =0; element < testSize;element++)
41     {
42         //Iterate to get average amount of time it takes to execute function.
43         for (int iteration =0; iteration < iterations; iteration++)
44         {
45             //Testing best case.
46             //Set the current vector.
47             usedVector = bestVector;
48             usedVector.resize(element);
49             startTime = high_resolution_clock::now();
50             func(usedVector);
51             endTime = high_resolution_clock::now();
52             duration = (endTime - startTime);
53             bestDuration += duration.count();
54
55             //Testing worst case.

```

```

56     ///Set the current vector.
57     usedVector = worstVector;
58     usedVector.resize(element);
59     startTime = high_resolution_clock::now();
60     func(usedVector);
61     endTime = high_resolution_clock::now();
62     duration = (endTime - startTime);
63     worstDuration += duration.count();
64
65     ///Testing average case.
66     ///Set the current vector.
67     usedVector = averageVector;
68     usedVector.resize(element);
69     startTime = high_resolution_clock::now();
70     func(usedVector);
71     endTime = high_resolution_clock::now();
72     duration = (endTime - startTime);
73     averageDuration += duration.count();
74 }
75 ///Get average time.
76 bestDuration /= iterations;
77 worstDuration /= iterations;
78 averageDuration /= iterations;
79
80 ///Write duration on file.
81 fileText += std::to_string(element);
82 fileText += ", ";
83 fileText += std::to_string(bestDuration);
84 fileText += ", ";
85 fileText += std::to_string(worstDuration);
86 fileText += ", ";
87 fileText += std::to_string(averageDuration);
88 fileText += "\n";
89 }
90 file.open(fileName);
91 file.clear();
92 file << fileText;
93 file.close();
94
95 }

```

---

## Benchmark Results

Nota: El eje X representa el numero de elementos en el vector mientras que el eje Y representa el tiempo(microsegundos).  
En estas graficas se observan los resultados de cada algoritmo utilizando diferentes entradas.

