Análisis y Algoritmos

Luis Alberto Pineda Chavez
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv17c.lpineda@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Mayo 23, 2019

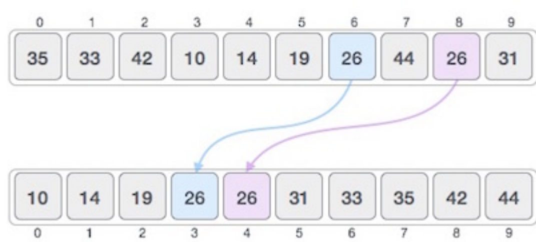
1) Sorting Algorithms

A sorting algorithm is made up of a series of instructions that takes an array as input, performs specified operations on the array, sometimes called a list, and outputs a sorted array.

Sorting algorithms are often taught early in computer science classes as they provide a straightforward way to introduce other key computer science topics like Big-O notation, divide-and-conquer methods, and data structures such as binary trees, and heaps. There are many factors to consider when choosing a sorting algorithm to use.

The objective

There are a lot of different algorithms to sort an array, all of them outputs an ascending order array, to solve the problem we are going to use two, Quick sort and Merge sort.



This the Merge Sort algorithm implementation:

```
//This method exist to simplify, inserting the first and last indices of the vector
void MergeSort(vector<unsigned int>& values)
{
        MergeSortHelper(values, 0, values.size() -1);
}

//This is the recursive method to divide the original vector and sort it
void MergeSortHelper(vector<unsigned int>& values, int left, int right)
{
        if (left < right)
        {
                unsigned int middle = left + (right - left) / 2;

                //Sort first and second halves
                MergeSortHelper(values, left, middle);
                MergeSortHelper(values, middle + 1, right);

                Merge(values, left, middle, right);
        }
}
```

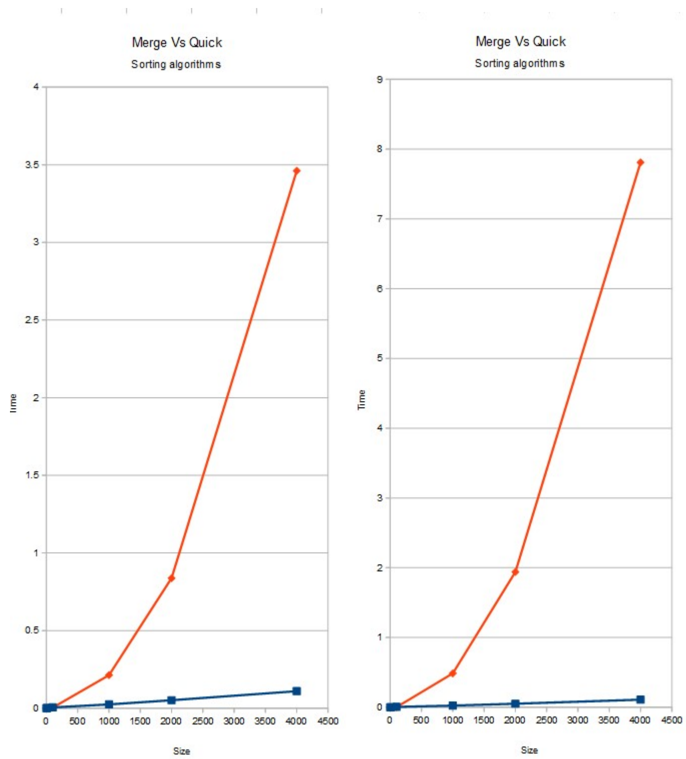
```
//This method merges a sorted version of the children
void Merge(vector<unsigned int>& values, int left, int middle, int right)
{

	//Set children limits
	int leftChild = middle - left + 1;
	int rightChild = right - middle;

	//Temporal child vectors
	vector <unsigned int> LeftChild(leftChild);
	vector <unsigned int> RightChild(rightChild);

	//Copy orginal data to children
	int i, j;
	for (i = 0; i < leftChild; i++)
	{
		LeftChild[i] = values[left + i];
	}
	for (j = 0; j < rightChild; j++)
	{
		RightChild[j] = values[middle + 1 + j];
	}

	//Sort children values
	i = 0;	j = 0;
	while (i < leftChild && j < rightChild)
	{
		if (LeftChild[i] <= RightChild[j])
		{
			values[left] = LeftChild[i];
			i++;
		}
		else
		{
			values[left] = RightChild[j];
			j++;
		}

		left++;
	}

	//Copy sorted childs into the original vector
	while (i < leftChild)
	{
		values[left] = LeftChild[i];
		i++;
		left++;
	}
	while (j < rightChild)
	{
		values[left] = RightChild[j];
		j++;
		left++;
	}
}
```

This the Quick Sort algorithm implementation:

```
//This method exist to simplify, inserting the first and last indices of the vector
void QuickSort(vector<unsigned int>& values)
{
	QuickSortHelper(values, 0, values.size() - 1);
}

//This is the recursive method that calculates the new pivot and calls the swapper
void QuickSortHelper(vector<unsigned int>& values, int low, int high)
{
	if (low < high)
	{
		//Position for the new vector partition
		int position = QuickSortSwapper(values, low, high);

		//Separately sort both sides of the vector
		QuickSortHelper(values, low, position - 1);
		QuickSortHelper(values, position + 1, high);
	}
}
```

```
//This method is the one that sorts the vector and returns the new pivot
unsigned int QuickSortSwapper(vector<unsigned int>& values, int low, int high)
{
        int pivot = values[high]; //Pivot is at the end of the current vector child

        int Low = (low - 1);  //Low position in the vector child

        for (int j = low; j < high; j++)
        {
                if (values[j] <= pivot) //If current element is smaller or equal to pivot
                {
                        Low++; //Increment Low element index
                        swap(values[Low], values[j]);
                }
        }

        swap(values[Low + 1], values[high]);

        return (Low + 1);
}
```

This graphics represents how much time it takes to each implementation sort the input:



Right is best case, left is worst case. Merge Sort's complexity is O(n * log n) for all situations, quick sort's best case shares the same, but the worst case has an O(n^2)$complexity$.

2) Seach Algorithms

A search algorithm is a sequence of steps to find an object or number inside an array, tree, matrix, list or any data structure, but for this practice we will stick to arrays/vectors.

Note: For this implementations to work properly, the arrays must by sorted.

The objective

To solve this problem, we will use two kinds of search algorithms, the linear and the binary implementations, both shares as input the value to search and the output is the position where that value is located.

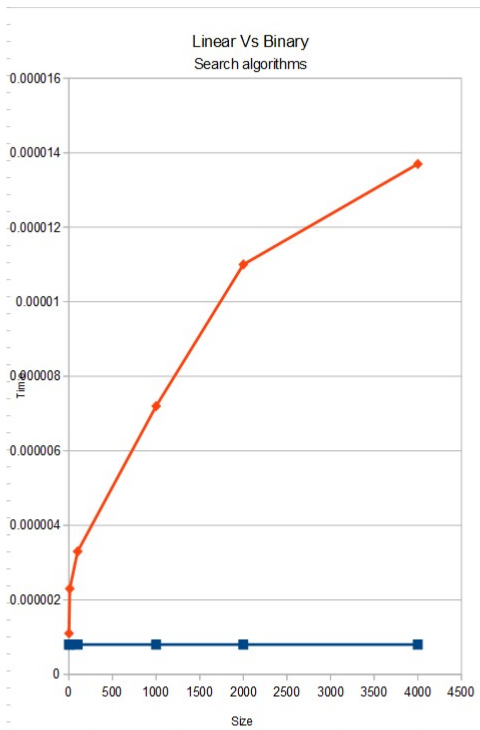
This the Linear search algorithm implementation:

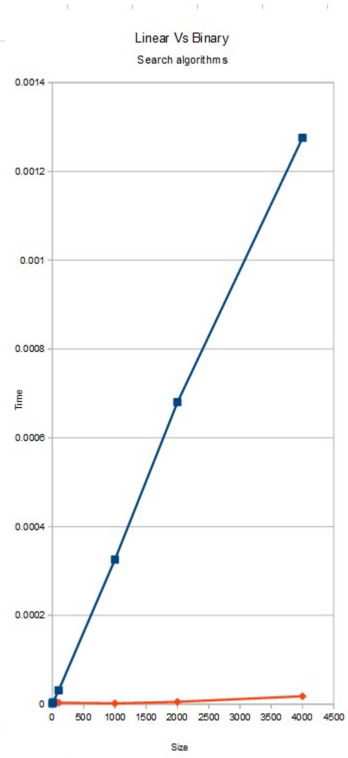
```
int LinearSearch(vector<unsigned int>& values, unsigned int number)
{
	//This compares every index of the array with the input number
	for (int i = 0; i < values.size(); i++)
	{
		//If the index of the array matches number to search
		if (values[i] == number)
		{
			//Return the index
			return i;
		}
	}

	//Returns an invalid index if the number wasn't find
	return -1;
}
```

This the Binary search algorithm implementation:

```
//This method exist to simplify, inserting the first and last indices of the vector
int BinarySearch(std::vector<unsigned int>& values, int number)
{
	return BinarySearchHelper(values, 0, values.size() - 1, number);
}


int BinarySearchHelper(vector<unsigned int>& values, int first, int last, int number)
{
	//Divide the array until we have just one element on each child
	if (last >= first)
	{
		//Calculate the middle of the aaray in order to split it
		int middle = first + (last - first) / 2;

		//Check if the middle of the arraya has the number we are looking for
		if (values[middle] == number)
		{
			//If it does, returns the index
			return middle;
		}

		//If not, call again the algorithm for both halves of the array
		if (values[middle] > number)
		{
			return BinarySearchHelper(values, first, middle - 1, number);
		}

		return BinarySearchHelper(values, middle + 1, last, number);
	}

	//Returns an invalid index if the number wasn't find
	return -1;
}
```

This graphics represents how much time it takes to each implementation sort the input:





Right is best case, left is worst case.

The linear search has a best case scenario of O(1) and a worst case of O(n), meanwhile binary search has an O(log n) complexity.

REFERENCIAS

[1] Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2009). Introduction to algorithms. Cambridge (England): Mit Press.