ITCS 6114 Project 1

Matthew Seman

Project 1 – Comparison Based Sorting Algorithms

Introduction

This report summarizes the implementation of various sorting algorithms in C++ by analyzing their time complexity. The sorting algorithms analyzed include insertion sort, merge sort, heap sort, quick sort with a pivot at the last element, and a median of three quick sort. Run time is experimentally measured against various arrays with sizes varying from 1000 to 60000 populated by random elements with values between 1 and 1000. The sorting algorithms are analyzed for three cases: unsorted arrays, sorted arrays in ascending order, and sorted arrays in descending order. Performance of the algorithms are visualized by plotting the time taken vs the size of the array. The pdPlots library (https://github.com/InductiveComputerScience/pbPlots) is used in order to plot the points and saved in .PNG files to the project folder.

Insertion Sort

Insertion sort works by using two indices i and j and a key value k. Index i increments up through the array and index j decrements down from each index i. The key value is set to the element's value at index i and each element prior to it is checked. If the values of the element at index j are greater than the key value then elements are moved through the array and the key value is placed at the sorted location. Insertion sort has a best case time analysis of O(n) and a worst case time analysis of $O(n^2)$. Below is my C++ implementation of insertion sort.

```
//Insertion sort
void insertionSort(int arr[], int arrSize)
{
	int i, j, k;
	for (i = 1; i < arrSize; i++) {
		k = arr[i];		//Set k equal to the element at index i
		j = i - 1;		//Index j behind i

		//If element value at j is greater than element value at i
		//then put value at index j at index i and decrement j
		while (j >= 0 && arr[j] > k) {
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		//Set value at index j + 1 = i to k
		arr[j + 1] = k;
	}
}
```

Merge Sort

Merge sort is a divide and conquer algorithm that takes an array and divides the array into two arrays from its center as long as the array contains more than one element. This continues

recursively until the array is divided into many arrays of single elements. These arrays are then merged together in sorted order one element at a time using the merge function. The merge sort algorithm works in linear time working in O(nlogn) for all cases since the array is split into 2 recursively. Below is my C++ implementation of merge sort using a separate merge function.

```
//Merge sort
void mergeSort(int arr[], int low, int high) {
	//If lowest index is greater than or equal to highest index return(Check
Array has more than 1 element)
	if (low >= high) {
		return;
	}

	//Find middle array index
	int mid = (low + high) / 2;
	//Recursively merge sort by splitting array from middle index
	mergeSort(arr, low, mid);
	mergeSort(arr, mid + 1, high);
	//Merge sorted arrays together
	merge(arr, low, mid, high);
}
```

```
//Merge two sub-arrays together
void merge(int arr[], int left, int mid, int right) {
	//Initialize number of elements of sub arrays left and right
	int sArrLeftSize = mid - left + 1;
	int sArrRightSize = right - mid;

	//Allocate memory for temporary array because C++ doesn't like array sizes
unless their
	//defined by constants or macros >:(
	int *leftArray = new int[sArrLeftSize], *rightArray = new int[sArrRightSize];

	//Store values from array into left and right arrays
	for (int i = 0; i < sArrLeftSize; i++) {
		leftArray[i] = arr[left + i];
	}
	for (int j = 0; j < sArrRightSize; j++) {
		rightArray[j] = arr[mid + 1 + j];
	}

	//Declare indeices of two sub array and main array that is the sub arrays
merged
	int indexSArrLeft = 0, indexSArrRight = 0, indexMergedArr = left;

	//index through left and right arrays and place them sorted into the merged
array
	//by comparing the elements at each index
	while (indexSArrLeft < sArrLeftSize && indexSArrRight < sArrRightSize) {
		if (leftArray[indexSArrLeft] <= rightArray[indexSArrRight]) {
			arr[indexMergedArr] = leftArray[indexSArrLeft];
			indexSArrLeft++;
		}
		else {
			arr[indexMergedArr] = rightArray[indexSArrRight];
```

```
                  indexSArrRight++;
            }
            indexMergedArr++;
      }
      //If there are any elements still in left array place them into merged array
      while (indexSArrLeft < sArrLeftSize) {
            arr[indexMergedArr] = leftArray[indexSArrLeft];
            indexSArrLeft++;
            indexMergedArr++;
      }
      //If there are any elements still in right array place them into merged array
      while (indexSArrRight < sArrRightSize) {
            arr[indexMergedArr] = rightArray[indexSArrRight];
            indexSArrRight++;
            indexMergedArr++;
      }
}
```

Heap Sort

Heap sort works by creating a max heap from the array selecting the largest of elements and replacing them with their nodes children is the children are greater than the node. If the node of the max heap is not the largest element in the heap then the largest is swapped with the node and the heap is max heapified recursively until the array is sorted. Heap sort also has a time complexity of O(nlogn) for all cases. Below is my implementation of heap sort in C++.

```
void heapSort(int arr[], int arrSize) {
      //Build the max heap using heapify function
      for (int i = arrSize / 2 - 1; i >= 0; i--) {
            heapify(arr, arrSize, i);
      }
      //Take elements from the max heap one at a time in sorted order
      for (int i = arrSize - 1; i > 0; i--) {
            //Swap elements at current root with last element in array
            std::swap(arr[0], arr[i]);
            //Max heapify the reduced array
            heapify(arr, i, 0);
      }
}
```

```
//Heapify a subtree as a Max heap
void heapify(int arr[], int arrSize, int node) {
      //Initialize largest as root node
      int largest = node;
      //initialize left child as index 2*node+1
      int left = 2 * node + 1;
      //initialize right child as index 2*node+2
      int right = 2 * node + 2;
      //If left child is larger than root node replace largest with left child
      if (left < arrSize && arr[left] > arr[largest]) {
            largest = left;
      }
      //If right child is larger than root node replace largest with right child
      if (right < arrSize && arr[right] > arr[largest]) {
            largest = right;
```

```
	}
	//If largest is not the root node swap node and largest elements and
recursively heapify
	if (largest != node) {
		std::swap(arr[node], arr[largest]);
		heapify(arr, arrSize, largest);
	}
}
```

Quick Sort

Quick sort works by choosing a pivot (In this case the last element) and partitioning the array into two arrays. The left array is populated by increment an index through the array and checking if the element at that index is less than the pivot. The right array is populated by decrementing an index starting at the second to last index (Skipping the pivot at the last index) and checking if the element at each index is greater than the pivot. The quicksort and partitioning are then done recursively until the entire array is sorted. The average time complexity of quick sort is O(nlogn) since on average the pivot will cause a near half-half partition. The worst time complexity is $O(n^2)$ if the pivot creates a partition of a single element every time. This will be the case if sorting an already sorted array and choosing the last or first element as the pivot. Below is my C++ implementation of quick sort.

```
void quickSort(int arr[], int low, int high) {
	//Check that the lowest index is less than the highest(Array is larger than 1
element)
	if (low < high) {
		//Choose partition index using partition function with last element as
pivot
		int partitionIndex = partitionLast(arr, low, high);

		//Recursively quicksort the partitioned arrays
		quickSort(arr, low, partitionIndex - 1);
		quickSort(arr, partitionIndex + 1, high);
	}
}
```

```
//Partition with last element in array as pivot
int partitionLast(int arr[], int low, int high) {
	int pivot = arr[high]; //Choose Last element as pivot

	int i = (low - 1);

	for (int j = low; j <= high; j++) {
		//If element at j is less than pivot value increment i by 1 then swap
with element at i
		if (arr[j] < pivot) {
			i++;
			std::swap(arr[i], arr[j]);
		}
	}
	std::swap(arr[i + 1], arr[high]);
```

```
	return (i + 1);
}
```

Modified Quick Sort

This modified version of quick sort uses a median of three approach to choosing the pivot value. When the number of elements for the array to be sorted is less than or equal to 15 then the function will use insertion sort instead. Median of three partition sorts the first, last, and middle elements of the array then swaps the middle element with the second to last element and chooses it as the pivot value. This can be implemented using the partition function choosing the last element as the pivot by ignoring the first and last element since they are already partitioned by the median of three sort. The average and worst case time complexities are still the same as the previous quicksort implementation, but using median of three to partition makes it much less likely to encounter worst case scenarios. Below is my implementation of the modified quick sort explained using C++.

```
//If the array to be quicksorted is less than or equal to 15 elements then use
insertion sort
//Modified quicksort using median of three partition
void modifiedQuicksort(int arr[], int low, int high) {
	if (high <= 15) {
		insertionSort(arr, high + 1);
	}
	else {
		//Standard quicksort except use partition pivot as median of first
middle and last elements
		if (low < high) {
			int partitionIndex = partitionMedianof3(arr, low, high);

			modifiedQuicksort(arr, low, partitionIndex - 1);
			modifiedQuicksort(arr, partitionIndex + 1, high);
		}
	}
}
```

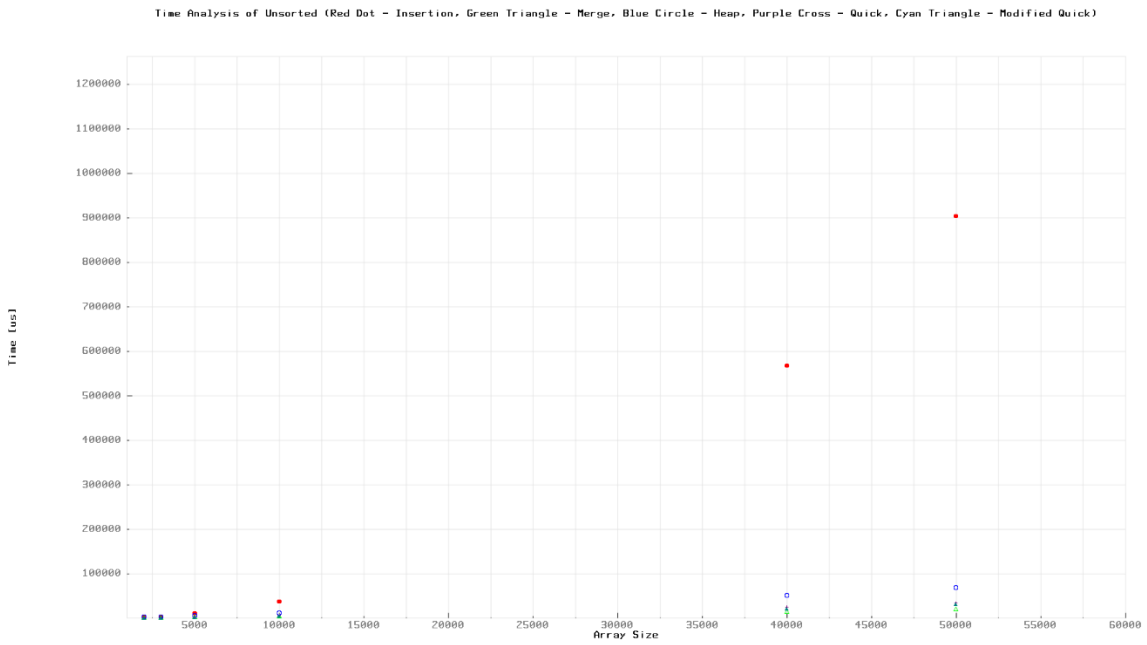
```
int partitionMedianof3(int arr[], int low, int high) {
	//Find middle element
	int mid = (high + low) / 2;

	//Sort elements at first, middle and last indices
	if (arr[mid] < arr[low]) {
		std::swap(arr[mid], arr[low]);
	}
	if (arr[high] < arr[low]) {
		std::swap(arr[high], arr[low]);
	}
	if (arr[high] < arr[mid]) {
		std::swap(arr[high], arr[mid]);
	}

	//Swap middle and second to last element
```

```
        std::swap(arr[mid], arr[high-1]);

        //Set pivot equal to second to last element
        //int pivot = arr[high-1];

        //Partition using second to last element as pivot
        //Ignore first and last elements since they are prepartitioned
        return partitionLast(arr, low + 1, high - 1);
}
```

Results

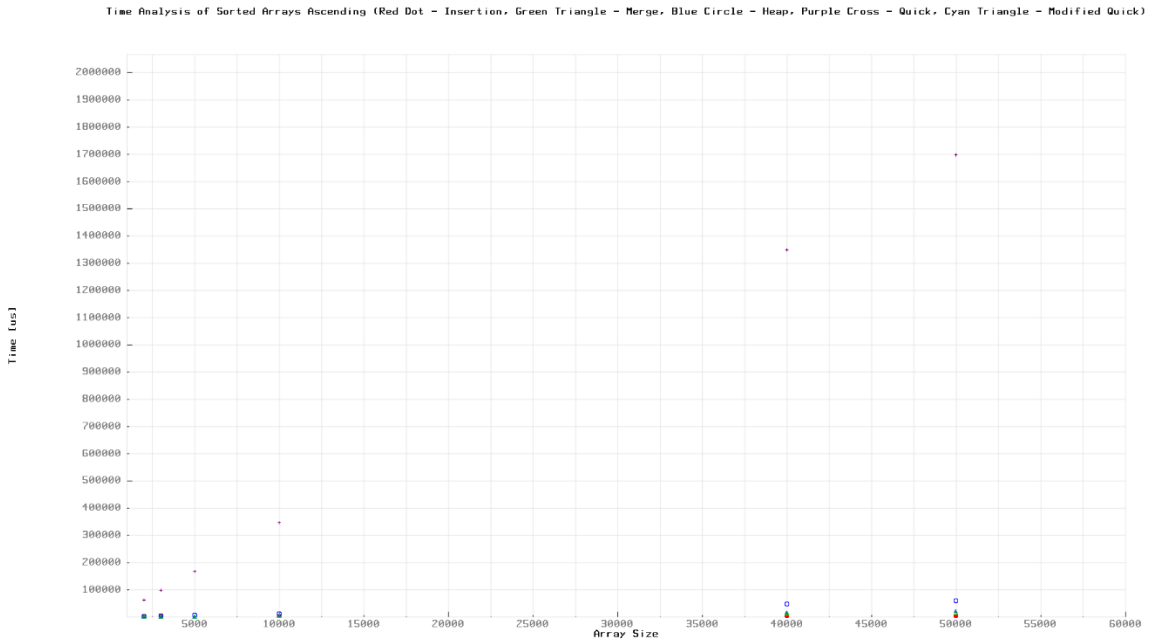
Below is the command prompt output of the time taken for each sorting algorithm for each array size. The outputs are separated by unsorted array, sorted arrays in ascending order and sorted arrays in descending order. After each command prompt output is the plot created by pbPlots for each respective type of array to be sorted

```
Time analysis of sorting unsorted arrays:
Time taken for insertion sort with array of size 1000: 636 microseconds.
Time taken for merge sort with array of size 1000: 527 microseconds.
Time taken for heap sort with array of size 1000: 934 microseconds.
Time taken for quick sort with array of size 1000: 606 microseconds.
Time taken for modified quick sort with array of size 1000: 605 microseconds.
Time taken for insertion sort with array of size 2000: 1848 microseconds.
Time taken for merge sort with array of size 2000: 905 microseconds.
Time taken for heap sort with array of size 2000: 2010 microseconds.
Time taken for quick sort with array of size 2000: 1225 microseconds.
Time taken for modified quick sort with array of size 2000: 1228 microseconds.
Time taken for insertion sort with array of size 3000: 3344 microseconds.
Time taken for merge sort with array of size 3000: 1227 microseconds.
Time taken for heap sort with array of size 3000: 3077 microseconds.
Time taken for quick sort with array of size 3000: 1962 microseconds.
Time taken for modified quick sort with array of size 3000: 1806 microseconds.
Time taken for insertion sort with array of size 5000: 9733 microseconds.
Time taken for merge sort with array of size 5000: 2250 microseconds.
Time taken for heap sort with array of size 5000: 5271 microseconds.
Time taken for quick sort with array of size 5000: 3245 microseconds.
Time taken for modified quick sort with array of size 5000: 2815 microseconds.
Time taken for insertion sort with array of size 10000: 36098 microseconds.
Time taken for merge sort with array of size 10000: 3741 microseconds.
Time taken for heap sort with array of size 10000: 11586 microseconds.
Time taken for quick sort with array of size 10000: 5887 microseconds.
Time taken for modified quick sort with array of size 10000: 6071 microseconds.
Time taken for insertion sort with array of size 40000: 566162 microseconds.
Time taken for merge sort with array of size 40000: 14946 microseconds.
Time taken for heap sort with array of size 40000: 50718 microseconds.
Time taken for quick sort with array of size 40000: 24576 microseconds.
Time taken for modified quick sort with array of size 40000: 21133 microseconds.
Time taken for insertion sort with array of size 50000: 902960 microseconds.
Time taken for merge sort with array of size 50000: 19564 microseconds.
Time taken for heap sort with array of size 50000: 68235 microseconds.
Time taken for quick sort with array of size 50000: 32734 microseconds.
Time taken for modified quick sort with array of size 50000: 31457 microseconds.
Time taken for insertion sort with array of size 60000: 1262783 microseconds.
Time taken for merge sort with array of size 60000: 22641 microseconds.
Time taken for heap sort with array of size 60000: 78527 microseconds.
Time taken for quick sort with array of size 60000: 36689 microseconds.
Time taken for modified quick sort with array of size 60000: 31630 microseconds.
```




```
Time analysis of sorting sorted arrays in ascending order:
Time taken for insertion sort with array of size 1000: 330 microseconds.
Time taken for merge sort with array of size 1000: 375 microseconds.
Time taken for heap sort with array of size 1000: 928 microseconds.
Time taken for quick sort with array of size 1000: 21398 microseconds.
Time taken for modified quick sort with array of size 1000: 524 microseconds.
Time taken for insertion sort with array of size 2000: 148 microseconds.
Time taken for merge sort with array of size 2000: 702 microseconds.
Time taken for heap sort with array of size 2000: 1936 microseconds.
Time taken for quick sort with array of size 2000: 60873 microseconds.
Time taken for modified quick sort with array of size 2000: 1163 microseconds.
Time taken for insertion sort with array of size 3000: 147 microseconds.
Time taken for merge sort with array of size 3000: 1289 microseconds.
Time taken for heap sort with array of size 3000: 2984 microseconds.
Time taken for quick sort with array of size 3000: 97578 microseconds.
Time taken for modified quick sort with array of size 3000: 1559 microseconds.
Time taken for insertion sort with array of size 5000: 119 microseconds.
Time taken for merge sort with array of size 5000: 1501 microseconds.
Time taken for heap sort with array of size 5000: 5084 microseconds.
Time taken for quick sort with array of size 5000: 167077 microseconds.
Time taken for modified quick sort with array of size 5000: 2474 microseconds.
Time taken for insertion sort with array of size 10000: 791 microseconds.
Time taken for merge sort with array of size 10000: 3309 microseconds.
Time taken for heap sort with array of size 10000: 11071 microseconds.
Time taken for quick sort with array of size 10000: 346754 microseconds.
Time taken for modified quick sort with array of size 10000: 5307 microseconds.
Time taken for insertion sort with array of size 40000: 1028 microseconds.
Time taken for merge sort with array of size 40000: 12451 microseconds.
Time taken for heap sort with array of size 40000: 46750 microseconds.
Time taken for quick sort with array of size 40000: 1347572 microseconds.
Time taken for modified quick sort with array of size 40000: 18051 microseconds.
Time taken for insertion sort with array of size 50000: 1023 microseconds.
Time taken for merge sort with array of size 50000: 14949 microseconds.
Time taken for heap sort with array of size 50000: 58274 microseconds.
Time taken for quick sort with array of size 50000: 1697341 microseconds.
Time taken for modified quick sort with array of size 50000: 22073 microseconds.
```

```
Time taken for insertion sort with array of size 60000: 1371 microseconds.
Time taken for merge sort with array of size 60000: 19543 microseconds.
Time taken for heap sort with array of size 60000: 72570 microseconds.
Time taken for quick sort with array of size 60000: 2066523 microseconds.
Time taken for modified quick sort with array of size 60000: 29266 microseconds.
```




```
Time analysis of sorting sorted arrays in descending order:
Time taken for insertion sort with array of size 1000: 1223 microseconds.
Time taken for merge sort with array of size 1000: 416 microseconds.
Time taken for heap sort with array of size 1000: 1609 microseconds.
Time taken for quick sort with array of size 1000: 13359 microseconds.
Time taken for modified quick sort with array of size 1000: 1071 microseconds.
Time taken for insertion sort with array of size 2000: 2989 microseconds.
Time taken for merge sort with array of size 2000: 742 microseconds.
Time taken for heap sort with array of size 2000: 1733 microseconds.
Time taken for quick sort with array of size 2000: 43109 microseconds.
Time taken for modified quick sort with array of size 2000: 1142 microseconds.
Time taken for insertion sort with array of size 3000: 6709 microseconds.
Time taken for merge sort with array of size 3000: 1133 microseconds.
Time taken for heap sort with array of size 3000: 2620 microseconds.
Time taken for quick sort with array of size 3000: 75717 microseconds.
Time taken for modified quick sort with array of size 3000: 1626 microseconds.
Time taken for insertion sort with array of size 5000: 17777 microseconds.
Time taken for merge sort with array of size 5000: 1912 microseconds.
Time taken for heap sort with array of size 5000: 4712 microseconds.
Time taken for quick sort with array of size 5000: 141970 microseconds.
Time taken for modified quick sort with array of size 5000: 2935 microseconds.
Time taken for insertion sort with array of size 10000: 68693 microseconds.
Time taken for merge sort with array of size 10000: 3480 microseconds.
Time taken for heap sort with array of size 10000: 9978 microseconds.
Time taken for quick sort with array of size 10000: 307828 microseconds.
Time taken for modified quick sort with array of size 10000: 4661 microseconds.
Time taken for insertion sort with array of size 40000: 1110950 microseconds.
Time taken for merge sort with array of size 40000: 13010 microseconds.
Time taken for heap sort with array of size 40000: 48091 microseconds.
Time taken for quick sort with array of size 40000: 1346387 microseconds.
Time taken for modified quick sort with array of size 40000: 20274 microseconds.
```

```
Time taken for insertion sort with array of size 50000: 1717084 microseconds.
Time taken for merge sort with array of size 50000: 15965 microseconds.
Time taken for heap sort with array of size 50000: 59686 microseconds.
Time taken for quick sort with array of size 50000: 1686932 microseconds.
Time taken for modified quick sort with array of size 50000: 24686 microseconds.
Time taken for insertion sort with array of size 60000: 2474286 microseconds.
Time taken for merge sort with array of size 60000: 19563 microseconds.
Time taken for heap sort with array of size 60000: 73420 microseconds.
Time taken for quick sort with array of size 60000: 2032378 microseconds.
Time taken for modified quick sort with array of size 60000: 28925 microseconds.
```