Asymptotic Notation and Sorting Project

TJ Maynes

October 15, 2014

1 Theory

In this report, we will be discussing three sorting algorithms and their corresponding asymptotic notations. The three sorting algorithms we will be testing include insertion sort, quicksort, and mergesort. We will be using the C++ programming language to implement and test these sorting algorithms with instances of various size and using four various types of arrays. The instance sizes we will be testing with includes: 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000. As required, we will be testing these sorting algorithms with four various types of arrays: an increasing order of arrays, a decreasing order of arrays, a random order of arrays, and an array consisting only of items of the same value.

Assuming the sorting algorithms are setup correctly, we will see from our tests a similiar realistic and theoritical time complexity. First, we will describe how each one of the sorting algorithms should be implemented and understanding their theoritical computational time complexity.

1.1 Insertion Sort

Insertion sort is a sorting method that starts at a single element in an array and then increments the rest of the elements in an array. Pseudocode for implementing an insertion sort is provided in Algorithm 1.

Algorithm 1: `Insertionsort`$(A, low, high)$

```
1 for i = 1 to n do
2 |   j = i
3 |   while j > 0 and A[j] < A[j − 1] do
4 |   |   Swap(A[j], A[j − 1])
5 |   |   j− = 1
6 |   end
7 end
```

Each iteration of the insertion sorting algorithm takes one input element, finds the index location in the array it needs to be set, and places the element in that index location. Insertion sort starts at the first element and loops through the array n times as shown in line 1 in Algorithm 1. Inside the outermost for-loop, we need to set the count of i to a new counter j as shown in line 2. The j counter will be used to run a while-loop until the j-th index of the array is greater than zero and the j-th index is less than the j-1 index of the array as shown in line 3 in Algorithm 1. We want to make sure that we only need to be inside the while-loop when the j-th index is less than the j-1 index

since we want to sort the array in increasing order (sorted order). Inside the while-loop we will be swapping the j-th index in place with the location of the j-1 index of the array as shown in line 4. The j element is decremented before finishing the while-loop as shown on line 5.

The best-case time complexity for insertion sort is $\theta(n)$ time, *if the inputs are already in sorted order*. If we already have inputs in sorted order then insertion sort just has to compare the first element to the next element which takes $\theta(n)$ time. The average-case and worst-case time complexity for insertion sort is $O(n^2)$, which occurs on some inputs where the algorithm would have to search through the unsorted array in $O(n)$ time. Once inside that inner-loop, we would have to shift the already sorted elements before inserting a new element back into the array which would take $O(n)$ time.

1.2 Quicksort

Quicksort is known as a divide and conquer algorithm which means we keep dividing a sorting problem down until we have simplier sub-problems that will be combined to give a solution to our sorting problem. Pseudocode for implementing a quicksort is provided in Algorithm 2.

Algorithm 2: `Quicksort`$(A, low, high)$

```
1 if low < high then
2 |  pivot − location = Partition(a, low, high)
3 |  Quicksort(A, low, pivot − location − 1)
4 |  Quicksort(A, pivot − location + 1, high)
5 end
```

For quicksort we will be using a recursive approach to sorting which involves partitioning the elements in the array into two groups. When running a quicksort algorithm we want to check if the lowest value in the array is smaller than the highest value in the array which is shown in line 1 in Algorithm 2. The partition algorithm will be instantiated by variable pivot (pivot minus location) as seen on line 2. This will be accomplish as seen on lines 3 and 4. Line 3 will recursively call back on the Quicksort algorithm to get elements starting at the lowest element to the pivot in the array. Line 4 will recursively call back on the Quicksort algorithm to get elements from the pivot location in the array to the highest element.

Pseudocode for implementing the partition portion of the Quicksort algorithm is provided in Algorithm 3.

Algorithm 3: `Partition`$(A, low, high)$

```
1 pivot = A[low]
2 leftwall = low
3 for i = low + 1 to high do
4 |  if A[i] < pivot then
5 |  |  ++leftwall
6 |  |  Swap(A[i], A[leftwall])
7 |  end
8 end
9 Swap(A[low], A[leftwall])
```

The partition algorithm (Algorithm 3) takes in three inputs: the array, a low input, and a high input. The pivot will equal the index value of low in array A as seen in line 1 in Algorithm 3. Next, we want

to instantiate a leftwall variable that equals our low input as seen on line 2. The leftwall variable will be used to help create the three sections of the array we want to partition. Next we will be looping through each array index i from low to high as shown in line 3. Inside the for-loop we will be checking to see if the array index is less than the pivot. If this is true we will be incrementing leftwall and swapping the i-th array index with the leftwall index of the array as seen in lines 4 to 7. If this is not true we will keep looping until the i-th index is no longer less than the high. Finally we will swap the i-th element of the array with the leftwall element of the array as shown in line 9. This will finish the partition algorithm.

The best-case and average-case time complexity for quicksort is $\theta(nlogn)$ time if we have a pivot that is as close to the median (of the inputs) as possible. However, if our pivot is not chosen carefully (such as far from the median or smallest element in the array), then quicksort would have a worst-case time complexity of $O(n^2)$.

1.3 Merge Sort

Mergesort is another example of a divide and conquer algorithm. Pseudocode for implementing a mergesort is provided in Algorithm 4.

Algorithm 4: `Mergesort`$(A, low, high)$

```
1 if low < high then
2 |   middle = (low − high)/2
3 |   Mergesort(A, low, middle)
4 |   Mergesort(A, middle + 1, high)
5 |   Merge(A, low, middle, high)
6 end
```

The merge sorting algorithm will be taking a recursive approach to sorting which involves continuously halving the number of elements at each level until we finally merge our elements back together (in sorted order). In our mergesort algorithm, we will have three inputs being passed into the algorithm: the array, the lowest element, and the highest element. In line 1 of Algorithm 4, we will be checking if low is less than high. If that condition is met, we will instantiate a middle variable which will take our range (high - low) and be divided by 2 as shown in line 2. Next, we will make a recursive call to the Mergesort algorithm so that we can only obtain elements between the low and middleelements as shown in line 3. Next, we will make a recursive call to the Mergesort algorithm so that we can only obtain elements between the middle and the highest element. Finally, our Merge algorithm will merge all the elements together (into array) the low to middle elements and the middle to high elements.

The best-case, worst-case, and average-case of mergesort is $O(nlogn)$ since we are halving number of elements at each level it takes $O(logn)$ time and $O(n)$ time to perform the merging on all elements per level.

Now that we have a good understanding of what each of the sorting algorithms does in theory, lets see the testing results from implementing these sorting algorithms against various arrays and instance sizes in C++.

2 Testing

The requirements for implementing our sorting algorithms includes: each function must be generic (i.e., a function that takes as input an array of any type of item), must implement the respective algorithm as described in the text book (located in references section), and implemented in file sorting.h. We may also use helper functions, which proves necessary to properly implement the sorting algorithms from the text book. Finally, we must give a tight upper and a tight lower bound for best case, average case, and worst case for each of the sorting algorithms.

The requirements for generating our arrays with specific properties, we must make sure that each generation function is generic and implemented in the file arrayGeneration.h.

The purpose of these testing parameters is to show see just how different each sorting algorithm's asymptotic time complexity will be when using a different set of arrays (increasing, decreasing, random, and constant) with various instance sizes (10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000).

The testing file, main.cpp, looped through each one of the sorting algorithms one at a time for each set of array tests. To add detail, the testing file runs three functions: main, testing, and printAverages. The main function begins our testing program, where we instantiate our starting test minimum and start size values as well looping through a string array of sorting algorithms to be processed by the testing function. For instance when insertionsort is called, the string value is passed into the testing function (along with minimum size and start size), once insertion sort testing is complete we iterate to the next algorithm (quicksort).

The testing function tests the respective sorting algorithm with various instance sizes and arrays. The testing function is set up to record and display the cpu time of each test as well as compute an average cpu time from that instance size's five runs. When the algorithm finishes running its respective instance sizes and arrays, we send each of our respective averages per array to a printAverages function. The testing function takes $\theta(n^2)$ time to complete since we have are running a very strict setup of different array tests to different instance sizes. The printAverages function prints all the average cpu times of the respective algorithm's specific array (increasing array, random array, etc). The printAverages function takes a total time of $\theta(n)$ to run through n size of each specific array.

Let us take a look at an increasing array on insertion sort, quicksort, and mergesort.

2.1 Increasing Array

Below is a table of CPU times (in milliseconds) from running an increasing array of various instance sizes on insertion sort.

	Insertion Sort Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	0.061	0.117	0.172	0.237	0.288	0.360	0.401	0.461	0.516	0.577
2	0.058	0.116	0.172	0.230	0.288	0.344	0.401	0.459	0.519	0.578
3	0.058	0.116	0.171	0.228	0.286	0.348	0.401	0.457	0.515	0.596
4	0.058	0.115	0.173	0.228	0.492	0.344	0.421	0.457	0.514	0.572
5	0.059	0.116	0.171	0.229	0.311	0.376	0.403	0.458	0.514	0.574
Avg	0.059	0.116	0.172	0.230	0.333	0.354	0.405	0.458	0.516	0.579

Below is a table of CPU times (in milliseconds) from running an increasing array of various instance sizes on quicksort (with pivot at last element).

Quicksort (with pivot at last element)

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	321.72	1300.73	2925.95	5203.45	8108.03	11682.94	15904.38	20793.47	27091.58	32408.57
2	324.92	1304.57	2918.97	5204.60	8138.04	11693.82	15856.76	20753.92	26253.95	32472.83
3	330.04	1298.75	2918.78	5212.22	8082.68	11706.75	15906.94	20708.09	26260.35	32303.35
4	325.56	1295.61	2926.33	5196.92	8093.69	11662.72	15849.08	20719.48	26257.79	32452.73
5	325.56	1291.07	2931.20	5181.50	8131.52	11655.42	15816.83	20929.40	26263.93	32408.70
Avg	325.56	1298.15	2924.25	5199.74	8110.79	11680.33	15866.80	20780.87	26425.52	32409.24

Below is a table of CPU times (in milliseconds) from running an increasing array of various instance sizes on quicksort (with pivot at random element).

Quicksort (with pivot at random element)

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	36.352	125.440	266.752	522.752	864.256	1134.080	1799.168	2408.960	3212.800	3970.816
2	28.160	150.016	371.712	508.160	662.784	869.120	1383.680	1312.256	2339.328	4060.928
3	35.840	98.560	314.880	348.672	851.200	1201.408	1490.944	2464.512	2767.616	2893.312
4	36.864	149.248	273.408	488.448	649.728	1374.464	1629.696	1736.448	2496.768	3818.240
5	40.448	100.608	270.592	560.384	664.832	1038.848	1815.296	1738.752	1942.528	3997.184
Avg	35.533	124.774	299.469	485.683	738.560	1123.584	1623.757	1932.186	2551.808	3748.096

Below is a table of CPU times (in milliseconds) from running an increasing array of various instance sizes on mergesort.

Mergesort

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	20.736	36.352	55.552	81.408	98.816	118.528	141.568	159.488	192.256	206.592
2	17.920	36.608	63.488	74.496	93.952	113.664	142.592	171.264	185.856	198.912
3	17.408	36.352	56.320	81.152	94.464	114.432	137.728	169.984	191.488	208.384
4	17.664	39.680	59.904	78.592	94.464	118.016	142.080	161.536	178.432	207.360
5	17.408	36.352	55.552	75.264	98.048	115.200	149.760	165.632	179.456	212.224
Avg	18.227	37.069	58.163	78.182	95.949	115.968	142.746	165.581	185.498	206.694

Let us take a look at running a decreasing array on insertion sort, quicksort, and mergesort.

2.2 Decreasing Array

Below is a table of CPU times (in milliseconds) from running an decreasing array of various instance sizes on insertion sort.

	Insertion Sort Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	343.48	1358.03	3208.37	5470.17	8511.99	12342.76	16726.17	21864.62	27644.57	34138.91
2	343.07	1406.07	3114.49	6532.65	8544.93	12298.99	16761.42	21907.02	27703.13	34064.54
3	346.01	1521.48	3126.88	5476.86	8556.75	12257.02	16721.43	21914.62	27759.48	34064.77
4	357.59	1530.58	3082.35	5470.10	8535.89	12319.88	16795.95	21858.43	27642.08	34189.05
5	346.22	1516.09	3083.02	5459.95	8558.20	12321.72	16773.84	21936.99	27716.12	34162.11
Avg	347.27	1466.45	3123.02	5681.95	8541.55	12308.07	16755.76	21896.34	27693.08	34123.87

Below is a table of CPU times (in milliseconds) from running an decreasing array of various instance sizes on quicksort (with pivot at last element).

	Quicksort (with pivot at last element) Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	225.02	920.70	2067.58	3646.46	5683.96	8342.65	11197.05	14647.04	18479.61	22827.26
2	228.60	924.54	2052.09	3650.04	5700.35	8273.02	11180.16	14627.58	18486.91	22841.98
3	226.81	915.58	2067.20	3650.17	5725.05	8230.27	11184.64	14575.48	18480.25	22825.98
4	230.01	907.00	2060.80	3651.84	5714.68	8249.98	11153.53	14618.49	18485.37	22820.09
5	227.07	904.57	2059.77	3652.86	5716.60	8203.52	11185.15	14605.82	18435.07	22820.47
Avg	227.50	914.48	2061.49	3650.27	5708.13	8259.89	11180.10	14614.88	18473.44	22827.16

Below is a table of CPU times (in milliseconds) from running an decreasing array of various instance sizes on quicksort (with pivot at random element).

	Quicksort (with pivot at random element) Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	32.512	170.496	225.024	651.264	715.776	1330.688	1914.368	2626.816	2246.144	1915.392
2	31.232	145.920	394.752	738.048	512.768	539.136	1311.744	2039.552	2552.064	2107.904
3	34.560	215.296	355.584	593.920	717.824	731.904	1972.480	2017.536	2618.368	4576.256
4	67.328	99.328	204.800	559.616	963.584	1131.264	897.024	1774.336	2541.568	3705.088
5	48.128	234.752	517.376	540.672	1236.736	1244.928	1237.760	2054.912	1729.024	2050.816
Avg	42.752	173.158	339.507	616.704	829.338	995.584	1466.675	2102.630	2337.434	2871.091

Below is a table of CPU times (in milliseconds) from running an decreasing array of various instance sizes on mergesort.

	Mergesort Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	20.992	41.216	55.552	74.752	94.720	114.432	141.312	154.880	179.200	195.584
2	21.248	37.120	55.296	78.592	94.976	118.016	140.544	154.880	178.432	199.168
3	17.920	36.096	55.296	75.52	97.024	116.992	147.712	157.952	174.336	195.584
4	18.688	40.448	58.112	80.128	97.792	114.432	135.936	154.880	184.064	199.424
5	18.944	36.096	55.296	75.264	96.512	114.432	137.984	155.136	181.248	197.376
Avg	19.558	38.195	55.910	76.851	96.205	115.661	140.698	155.546	179.456	197.427

Let us take a look at running a random array on insertion sort, quicksort, and mergesort.

2.3 Random Array

Below is a table of CPU times (in milliseconds) from running a random array of various instance sizes on insertion sort.

	Insertion Sort Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	161.152	657.344	1476.928	2635.584	4109.63	5889.15	8183.74	10531.58	13293.50	16410.11
2	164.416	663.808	1470.080	2648.832	4105.79	5902.27	8184.96	10486.14	13310.46	16437.82
3	167.488	659.392	1476.160	2620.864	4110.46	5905.40	8016.57	10567.23	13301.56	16438.33
4	161.856	650.688	1475.264	2714.304	4100.03	5897.79	8066.04	10476.22	13236.92	16430.97
5	163.264	652.160	1484.608	2632.896	4097.21	5891.84	8041.60	10490.56	13365.37	16333.63
Avg	163.635	656.678	1476.608	2650.496	4104.62	5897.29	8098.58	10510.34	13301.56	16410.17

Below is a table of CPU times (in milliseconds) from running a random array of various instance sizes on quicksort (with pivot at last element).

Quicksort (with pivot at last element)

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	1.408	2.816	4.352	6.016	8.704	9.472	11.520	13.312	14.464	16.128
2	1.280	2.816	4.480	6.144	7.808	9.472	11.136	13.184	17.152	16.512
3	1.280	2.816	4.480	6.144	7.936	9.344	10.752	13.440	15.360	16.256
4	1.280	2.816	4.608	6.016	7.552	9.344	11.392	12.800	14.720	17.024
5	1.408	2.816	4.480	8.576	7.808	9.472	11.520	12.672	14.720	16.000
Avg	1.331	2.816	4.480	6.579	7.962	9.421	11.264	13.082	15.283	16.384

Below is a table of CPU times (in milliseconds) from running an random array of various instance sizes on quicksort (with pivot at random element).

Quicksort (with pivot at random element)

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	5.376	12.032	23.808	33.024	38.912	57.856	70.400	130.048	149.248	129.280
2	4.608	15.616	22.784	35.072	41.216	75.776	85.248	88.320	111.616	122.368
3	5.376	9.216	27.136	29.440	39.168	87.040	99.584	93.696	132.608	91.136
4	5.888	13.568	26.624	40.448	36.608	54.016	80.384	103.424	98.816	157.696
5	4.352	9.984	19.456	38.656	50.176	43.008	71.680	84.480	103.936	134.400
Avg	5.120	12.083	23.962	35.328	41.216	63.539	81.459	99.994	119.245	126.976

Below is a table of CPU times (in milliseconds) from running a random array of various instance sizes on mergesort.

Mergesort

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	18.944	39.680	60.928	81.664	104.192	125.184	156.928	199.680	204.544	220.672
2	18.944	39.680	60.672	84.992	104.192	132.608	151.808	203.776	199.680	222.720
3	18.944	39.680	63.488	81.920	115.968	126.976	172.800	200.704	199.168	224.256
4	19.200	39.680	60.160	86.272	103.424	125.952	171.776	198.144	205.056	243.712
5	19.200	42.240	60.416	82.688	103.424	129.024	176.896	198.656	196.864	221.952
Avg	19.046	40.192	61.133	83.507	106.240	127.949	166.042	200.192	201.062	226.662

Let us take a look at a constant array on insertion sort, quicksort, and mergesort.

2.4 Constant Array

Below is a table of CPU times (in milliseconds) from running a constant array of various instance sizes on insertion sort.

Insertion Sort

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	0.064	0.128	0.256	0.896	0.768	0.832	0.768	1.088	1.024	1.344
2	0.128	0.192	0.256	0.512	0.512	0.576	0.576	0.704	0.896	0.960
3	0.128	0.128	0.256	0.512	0.448	0.576	0.640	0.640	1.408	0.960
4	0.064	0.192	0.320	0.512	0.512	0.512	0.704	0.704	1.152	0.960
5	0.064	0.192	0.512	0.384	0.512	0.512	0.640	0.640	0.960	0.896
Avg	0.090	0.166	0.320	0.563	0.550	0.602	0.666	0.755	1.088	1.024

Below is a table of CPU times (in milliseconds) from running a constant array of various instance sizes on quicksort (with pivot at last element).

Quicksort (with pivot at last element)

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	134.14	532.35	1191.29	2108.54	3305.72	4782.84	6484.60	8514.81	10717.69	13246.72
2	130.17	529.02	1182.97	2115.07	3311.61	4777.85	6519.29	8492.28	10731.26	13206.27
3	130.04	529.15	1186.81	2106.88	3307.52	4785.02	6465.92	8483.32	10726.40	13250.81
4	130.04	524.92	1187.20	2136.06	3309.69	4752.25	6496.89	8468.48	10773.24	13258.49
5	133.37	523.00	1194.24	2113.53	3320.57	4772.09	6494.59	8483.84	10729.72	13251.32
Avg	131.55	527.69	1188.50	2116.01	3311.02	4774.01	6492.26	8488.55	10735.66	13242.72

Below is a table of CPU times (in milliseconds) from running a constant array of various instance sizes on quicksort (with pivot at random element).

Quicksort (with pivot at random element)

	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	130.30	533.50	1191.42	2107.13	3391.74	4854.27	6752.00	8854.78	11144.96	14538.75
2	140.03	532.48	1192.70	2125.05	3354.62	4974.08	6795.77	8693.50	11687.68	14448.89
3	132.35	532.48	1180.16	2122.75	3318.27	4926.97	6644.48	8667.64	11654.40	14656.51
4	130.81	527.36	1181.44	2131.45	3331.07	4899.07	6688.25	8783.10	11851.77	14380.54
5	132.60	535.55	1205.24	2134.01	3340.80	5288.70	6575.87	8602.62	11763.71	13408.76
Avg	133.22	532.27	1190.19	2124.08	3347.30	4988.62	6691.27	8720.33	11620.50	14286.69

Below is a table of CPU times (in milliseconds) from running a constant array of various instance sizes on mergesort.

	Mergesort									
	Instance Size									
	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
1	19.968	34.560	56.064	77.312	91.392	109.312	130.304	155.136	174.848	189.184
2	16.640	34.816	52.736	80.384	94.720	112.896	136.448	156.160	176.640	187.392
3	16.896	34.560	56.064	77.056	96.256	112.128	130.560	162.048	176.384	193.024
4	19.712	34.816	53.248	72.448	91.904	123.136	132.096	148.992	172.288	197.120
5	16.640	34.560	60.928	75.520	91.648	109.312	139.520	148.736	177.920	192.512
Avg	17.971	34.662	55.808	76.544	93.184	113.357	133.786	154.214	175.616	191.846

2.5 Average Times

Below is a table of average CPU times (in milliseconds) for running insertion sort on our generated arrays and various instance sizes.

Insertion Sort

Instance Size	Instance Type			
	Increasing	Decreasing	Random	Constant
10000	0.059	347.278	163.635	0.090
20000	0.116	1466.453	656.678	0.166
30000	0.172	3123.025	1476.608	0.320
40000	0.230	5681.952	2650.496	0.563
50000	0.333	8541.556	4104.627	0.550
60000	0.354	12308.076	5897.293	0.602
70000	0.405	16755.766	8098.586	0.666
80000	0.458	21896.340	10510.349	0.755
90000	0.516	27693.082	13301.568	1.088
100000	0.579	34123.879	16410.176	1.024

Below is a table of average CPU times (in milliseconds) for running quicksort (with pivot at last element) on our generated arrays and various instance sizes.

Quicksort (pivot at last element)

Instance Size	Instance Type			
	Increasing	Decreasing	Random	Constant
10000	325.568	227.507	1.331	131.558
20000	1298.150	914.483	2.816	527.693
30000	2924.250	2061.491	4.480	1188.506
40000	5199.744	3650.279	6.579	2116.019
50000	8110.797	5708.134	7.962	3311.027
60000	11680.333	8259.892	9.421	4774.016
70000	15866.803	11180.108	11.264	6492.263
80000	20780.875	14614.888	13.082	8488.551
90000	26425.521	18473.445	15.283	10735.667
100000	32409.240	22827.160	16.384	13242.727

Below is a table of average CPU times (in milliseconds) for running quicksort (with pivot at random element) on our generated arrays and various instance sizes.

Quicksort (pivot at random element)

Instance Size	Instance Type			
	Increasing	Decreasing	Random	Constant
10000	35.533	42.752	5.120	133.222
20000	124.774	173.158	12.083	532.275
30000	299.469	339.507	23.962	1190.195
40000	485.683	616.704	35.328	2124.083
50000	738.560	829.338	41.216	3347.303
60000	1123.584	995.584	63.539	4988.621
70000	1623.757	1466.675	81.459	6691.276
80000	1932.186	2102.630	99.994	8720.333
90000	2551.808	2337.434	119.245	11620.506
100000	3748.096	2871.091	126.976	14286.693

Below is a table of average CPU times (in milliseconds) for running mergesort on our generated arrays and various instance sizes.

Mergesort

Instance Size	Instance Type			
	Increasing	Decreasing	Random	Constant
10000	18.227	19.558	19.046	17.971
20000	37.069	38.195	40.192	34.662
30000	58.163	55.910	61.133	55.808
40000	78.182	76.851	83.507	76.544
50000	95.949	96.205	106.240	93.184
60000	115.968	115.661	127.949	113.357
70000	142.746	140.698	166.042	133.786
80000	165.581	155.546	200.192	154.214
90000	185.498	179.456	201.062	175.616
100000	206.694	197.427	226.662	191.846

3 Matching Theory to Practice

In this section we will be examining the test results from the previous section to decide the time complexity of the best-case, average-case, or worst-case input instance of each sorting algorithm. First, we will discuss the insertion sort algorithm.

3.1 Insertion Sort

The expected runtime is figured by assuming that it takes the specified sorting algorithm 1 second to sort 1,000 items on your local machine.

	Expected Runtime	Actual Runtime
10000	0.01	0.059
20000	0.02	0.116
30000	0.03	0.172
40000	0.04	0.230
50000	0.05	0.333
60000	0.06	0.354
70000	0.07	0.405
80000	0.08	0.458
90000	0.09	0.516
100000	0.1	0.579

1. Increasing Array - Based on the results from average tests, Insertion sort is taking $\theta(n)$ time to run through the increasing array. Since the permutations in an increasing array are already sorted, then the algorithm takes $\theta(n)$ to check if the first element is smaller than n elements after it (which doesn't require n insertions). This only requires $\theta(n)$ time to sort through all the n elements of the already sorted array, which is insertion sort's best-case time.

	Expected Runtime	Actual Runtime
10000	0.1	347.278
20000	0.4	1466.453
30000	0.9	3123.025
40000	1.6	5681.952
50000	2.5	8541.556
60000	3.6	12308.076
70000	4.9	16755.766
80000	6.4	21896.340
90000	8.1	27693.082
100000	10.0	34123.879

2. Decreasing Array - Based on the results from average tests, Insertion sort is taking $\theta(n^2)$ time to run through the decreasing array. Since the permutations in a decreasing array are inversed, then the algorithm has to check n elements whether each n element after that n element is smaller than it. The problem is that n elements need to be shifted before being inserted, which also takes $\theta(n)$ time. This requires $\theta(n^2)$ to traverse through the decreasing array, which is insertion sort's worst-case time.

	Expected Runtime	Actual Runtime
10000	0.1	163.635
20000	0.4	656.678
30000	0.9	1476.608
40000	1.6	2650.496
50000	2.5	4104.627
60000	3.6	5897.293
70000	4.9	8098.586
80000	6.4	10510.349
90000	8.1	13301.568
100000	10.0	16410.176

3. Random Array - Based on the results from average tests, Insertion sort is taking $\theta(n^2)$ time to run through the random array. Since the permutations in a random array are randomly sorted, then the algorithm takes $\theta(n^2)$ time to check whether each n element is smaller than n elements after it. The same problem occurs when insertion sort is sorting through a decreasing array. This requires $\theta(n^2)$ to sort through the entire random array, which is insertion sort's average-case time.

	Expected Runtime	Actual Runtime
10000	0.01	0.090
20000	0.02	0.166
30000	0.03	0.320
40000	0.04	0.563
50000	0.05	0.550
60000	0.06	0.602
70000	0.07	0.666
80000	0.08	0.755
90000	0.09	1.088
100000	0.1	1.024

4. Constant Array - Based on the results from average tests, Insertion sort is taking $\theta(n)$ time to run through the constant array. Since the permutations in a constant array are already sorted (since all the values are the same), then the algorithm takes $\theta(n)$ time to check if the first element is smaller than n elements after it. This only requires $\theta(n)$ time to sort through all the n elements of an array of constant values, which is insertion sort's best-case time.

3.3 Quicksort (last element as pivot)

The expected runtime is figured by assuming that it takes the specified sorting algorithm 1 second to sort 1,000 items on your local machine.

	Expected Runtime	Actual Runtime
10000	0.1	325.568
20000	0.4	1298.150
30000	0.9	2924.250
40000	1.6	5199.744
50000	2.5	8110.797
60000	3.6	11680.333
70000	4.9	15866.803
80000	6.4	20780.875
90000	8.1	26425.521
100000	10.0	32409.240

1. Increasing Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(n^2)$ time to run through the increasing array. Since we chose to use the last element as our pivot and the permutations in an increasing array are already sorted, then the algorithm will not have any elements on the right side of the pivot (last element), which will cause each recursive call to quicksort to take one less element than the previous element (n - 1). This requires $\theta(n^2)$ time to sort through all the n elements in an increasing array, which is quicksort's worst-case time.

	Expected Runtime	Actual Runtime
10000	0.1	227.507
20000	0.4	1298.150
30000	0.9	914.483
40000	1.6	3650.279
50000	2.5	5708.134
60000	3.6	8259.892
70000	4.9	11180.108
80000	6.4	14614.888
90000	8.1	18473.445
100000	10.0	22827.160

2. Decreasing Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(n^2)$ time to run through the decreasing array. Since we chose the last element as our pivot and the permutations in a decreasing array are inversed, then the same scenario that occurred with an increasing array also will occur. This, like the increasing array input, requires $\theta(n^2)$ time to sort through all the n elements in a decreasing array, which is quicksort's worse-case time.

	Expected Runtime	Actual Runtime
10000	13.3	1.331
20000	26.6	2.816
30000	40.0	4.480
40000	53.3	6.579
50000	66.6	7.962
60000	80.0	9.421
70000	93.3	11.264
80000	106.6	13.082
90000	120.0	15.283
100000	166.6	16.384

3. Random Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(nlogn)$ time to run through the random array. Since we chose to use the last element as our pivot and the permutations in a random array are random, then the algorithm will have a better chance that our last element could be closer to a median value than the lowest or highest value (which would otherwise give us a worst-case time of $\theta(n^2)$). This requires $\theta(nlogn)$ time to sort through all n elements of a random array, which is quicksort's average-case time.

	Expected Runtime	Actual Runtime
10000	13.3	131.558
20000	26.6	527.693
30000	40.0	1188.506
40000	53.3	2116.019
50000	66.6	3311.027
60000	80.0	4774.016
70000	93.3	6492.263
80000	106.6	8488.551
90000	120.0	10735.667
100000	166.6	13242.727

4. Constant Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(nlogn)$ time to run through the constant array. Since we chose to use the last element as our pivot and the permutations in a constant array the same values, then the algorithm will have a better chance that our last element could be closer to a median value than the lowest or highest value (which would otherwise give us a worst-case time of $\theta(n^2)$) since all values are the same. This requires $\theta(nlogn)$ time to sort through all n elements of a random array, which is quicksort's average-case time.

3.2 Quicksort (random element as pivot)

The expected runtime is figured by assuming that it takes the specified sorting algorithm 1 second to sort 1,000 items on your local machine.

	Expected Runtime	Actual Runtime
10000	0.1	35.533
20000	0.4	124.774
30000	0.9	299.469
40000	1.6	485.683
50000	2.5	738.560
60000	3.6	1123.584
70000	4.9	1623.757
80000	6.4	1932.186
90000	8.1	2551.808
100000	10.0	3748.096

1. Increasing Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(n^2)$ time to run through the increasing array. Since our pivot was selected at a random element in our array and the permutations in an increasing array are already sorted, then it will have to sort every item to left or right of the pivot, which will cause each recursive call to quicksort to take one less element than the previous element (n - 1) i-th number of times. This will require $\theta(n^2)$ time to sort through the already sorted elements in the increasing array, where this will be quicksort's worst-case sorting time.

	Expected Runtime	Actual Runtime
10000	0.1	42.752
20000	0.4	173.158
30000	0.9	339.507
40000	1.6	616.704
50000	2.5	829.338
60000	3.6	995.584
70000	4.9	1466.675
80000	6.4	2102.630
90000	8.1	2337.434
100000	10.0	2871.091

2. Decreasing Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(n^2)$ time to run through the decreasing array. Since our pivot was selected at a random element in our array and the permutations in an decreasing array are inversely sorted, then it will have to sort every item to left or right of the pivot, which will cause each recursive call to quicksort to take one less element than the previous element (n - 1) i-th number of times. This will require $\theta(n^2)$ time to sort through the inversely sorted elements in the decreasing array, where this will be quicksort's worst-case sorting time.

	Expected Runtime	Actual Runtime
10000	13.3	5.120
20000	26.6	12.083
30000	40.0	23.962
40000	53.3	35.328
50000	66.6	41.216
60000	80.0	63.539
70000	93.3	81.459
80000	106.6	99.994
90000	120.0	119.245
100000	166.6	126.976

3. Random Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(nlogn)$ time to run through the random array. Since our pivot was selected by a random element in our array and the permutations in a random array are random, then we will have a better chance of not hitting its worst-case time of $\theta(n^2)$. The random pivot on a random array probabilistically minimizes our chances of getting closer to the median than if we were to pick the last element as our pivot. This will require $\theta(nlogn)$ time to sort through the randomly sorted elements in the random array, where this will be quicksort's average-case sorting time.
4. Constant Array - Based on the results from expected runtime to actual runtime averages, Quicksort is taking $\theta(n^2)$ time to run through the constant array. Since our pivot is selected by a random element in our array and the permutations in a constant array are the same value, then it will have to sort every item to left or right of the pivot, which will cause the algorithm to make each recursive call to quicksort to take one less element than the previous element

	Expected Runtime	Actual Runtime
10000	0.1	133.222
20000	0.4	532.275
30000	0.9	1190.195
40000	1.6	2124.083
50000	2.5	3347.303
60000	3.6	4988.621
70000	4.9	6691.276
80000	6.4	8720.333
90000	8.1	11620.506
100000	10.0	14286.693

(n - 1) i-th number of times. This will require $\theta(n^2)$ time to sort through the constant value elements in the constant array, where this will be quicksort's worst-case sorting time.

3.4 Mergesort

The expected runtime is figured by assuming that it takes the specified sorting algorithm 1 second to sort 1,000 items on your local machine.

	Expected Runtime	Actual Runtime
10000	13.3	18.227
20000	26.6	37.069
30000	40.0	58.163
40000	53.3	78.182
50000	66.6	95.949
60000	80.0	115.968
70000	93.3	142.746
80000	106.6	165.581
90000	120.0	185.498
100000	166.6	206.694

1. Increasing Array - Based on the results from expected runtime to actual runtime averages, Mergesort is taking $\theta(nlogn)$ time to run through the increasing array. Since we are always doubling the size (of the already sorted array) at each level taking $\theta(logn)$ time and always taking $\theta(n)$ time to merge all the elements on that level, we will have a best-case running time of $\theta(nlogn)$ (where all of the elements are already sorted).

	Expected Runtime	Actual Runtime
10000	13.3	19.558
20000	26.6	38.195
30000	40.0	55.910
40000	53.3	76.851
50000	66.6	96.205
60000	80.0	115.661
70000	93.3	140.698
80000	106.6	155.546
90000	120.0	179.456
100000	166.6	197.427

2. Decreasing Array - Based on the results from expected runtime to actual runtime averages, Mergesort is taking $\theta(nlogn)$ time to run through the decreasing array. Since we are always doubling the size (of the inversely sorted array) at each level taking $\theta(logn)$ time and always taking $\theta(n)$ time to merge all the elements on that level, we will have an average-case running time of $\theta(nlogn)$ (where all of the elements are inversely sorted).

	Expected Runtime	Actual Runtime
10000	13.3	19.046
20000	26.6	40.192
30000	40.0	61.133
40000	53.3	83.507
50000	66.6	106.240
60000	80.0	127.949
70000	93.3	166.042
80000	106.6	200.192
90000	120.0	201.062
100000	166.6	226.662

3. Random Array - Based on the results from expected runtime to actual runtime averages, Mergesort is taking $\theta(nlogn)$ time to run through the decreasing array. Since we are always doubling the size (of the randomly sorted array) at each level taking $\theta(logn)$ time and always taking $\theta(n)$ time to merge all the elements on that level, we will have a worst-case running time of $\theta(nlogn)$ (where all of the elements are randomly sorted).

	Expected Runtime	Actual Runtime
10000	13.3	17.971
20000	26.6	34.662
30000	40.0	55.808
40000	53.3	76.544
50000	66.6	93.184
60000	80.0	113.357
70000	93.3	133.786
80000	106.6	154.214
90000	120.0	175.616
100000	166.6	191.846

4. Constant Array - Based on the results from expected runtime to actual runtime averages, Mergesort is taking $\theta(nlogn)$ time to run through the constant array. Since we are always doubling the size (of the constant array) at each level taking $\theta(logn)$ time and always taking $\theta(n)$ time to merge all the elements on that level, we will have a best-case running time of $\theta(nlogn)$ (where all of the elements have a constant value).

4 Code

Note that the information presented in this section is also included in the README file.

Based on the requirements, the sorting algorithms are in sorting.h, the generated arrays are in arrayGeneration.h, and the testing functions are in main.cpp. Therefore, to compile simply run:

make clean
make

The executable will be named main.
For this code we use console output. The project description requires file output which means we must redirect the console output. Therefore, to run the program type:

./main > output.txt

Since this is not a simple command, a shell script is provided (test.sh). The script will compile and run the code. Note that you may have update the attributes for test.sh to make it an executable file. To ensure that test.sh is an executable file type:

chmod u+x test.sh

To compile and run the code type:
./test.sh

References

[1] cppreference.com. http://en.cppreference.com/w/, 2014.

[2] Insertion sort - wikipedia. http://en.wikipedia.org/wiki/Insertion_sort, 2014.

[3] Mergesort - wikipedia. http://en.wikipedia.org/wiki/Merge_sort, 2014.

[4] Quicksort - wikipedia. http://en.wikipedia.org/wiki/Quicksort, 2014.

[5] Steven S. Skiena. The algorithm design manual, 2008.