Quick Intro:

The following describes the analysis of quicksort, merge sort, and insertion sort. I will time each of the sorts and count how many basic operations are performed in each sorting. Each sort method will be performed on random, sorted, and reverse-sorted date 3 times. I will take averages and compare the expected numbers with the numbers I measured during runtime. These differences will be displayed both as charts and as line graphs.

The tests have been run on a system with the following specifications: (important for times and perhaps any compatibility issues with scripts, but not basic operation counts)

Kernel:	Linux 2.6.20-ARCH
Java version:	1.6.0
Bash version:	GNU bash, version 3.2.5(1)-release (i686-pc-linux-gnu)
Processor:	Intel(R) Pentium(R) M processor 1.50GHz
RAM:	512 MB

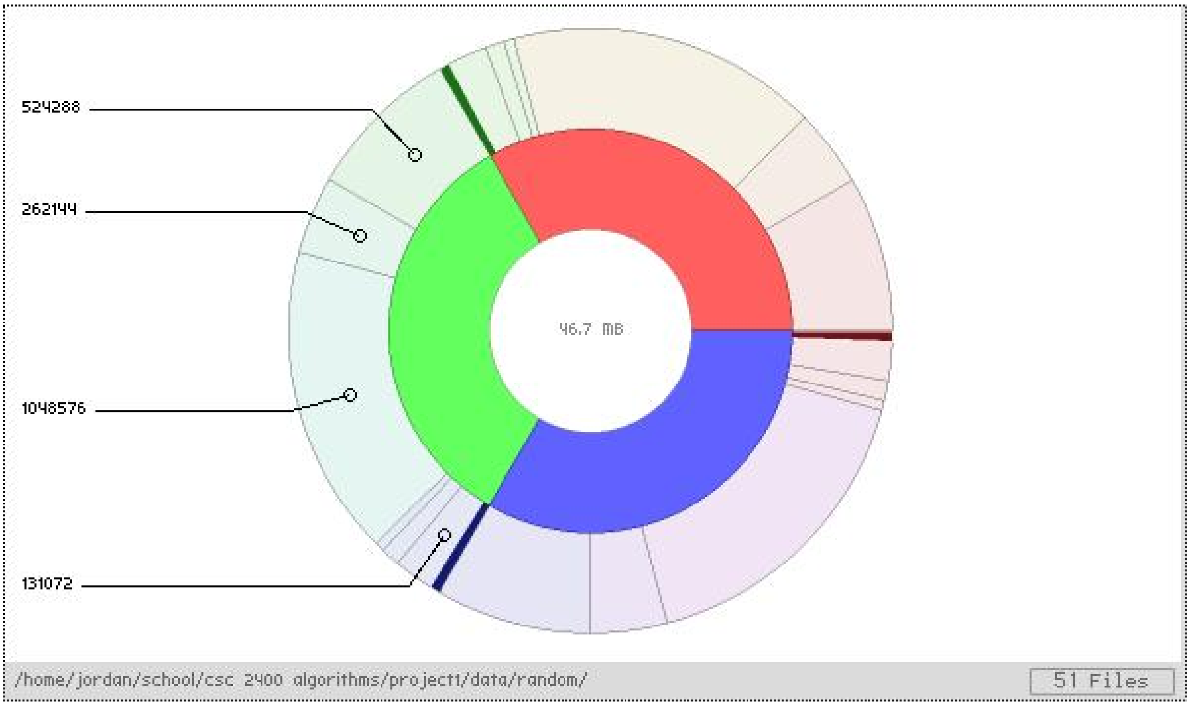
After the charts and tables are displayed, I will discuss the various algorithms and when it would be best to use each. Anything I find interesting or peculiar along the way will be discussed after the charts as well.

Data Generation:

The numbers to be used in the sorting algorithms were generated using Java and two classes named **getRandoms** and **RandomNumbers**. The number of numbers to generate is passed into **getRandoms**. Then, **getRandoms** calls **RandomNumbers** this many times. Each time a number is received, it is printed directly onto the console. I used a bash script called **generateInput** to direct these numbers to their proper locations. I had been using the c rand() function to generate random numbers, but I found that it gave me the same data every time, so I decided to switch to java, which I am more familiar with. However, the java implementation I used is quite slow, so data generation takes longer than the c implementation did.

When run, **generateInput** looks in the directory named "values/" to get the numbers to pass into **getRandoms**. The files in "values/" are simply empty files named 32, 64, 128, 256.....1048576. Inside the "data" directory I have put three folders: "random/" "reverse/" and "sorted/" Files of length obtained from the "values/" directory are put into each of these 3 folders. Additionally, the command **sort -n** is called on the files in the "sorted/" and **sort -nr** on the files in the"reverse/" directories when adding the randomly generated data.

The result looks like this, with each large colored section being one of the directories described above. Each directory has identically-named files but not identical contents of files:



Time Measurement

I measured time using Java's built in SimpleDateFormat class. I grabbed the time right before I started the sort and also immediately after the sort had completed. Rather than write another program to parse the times and calculate the running time I decided to do it manually by subtracting th start time from the end time. To record the times, I redirected the output of the Java sorting command to a file in the results directory. For each subsequent test of a method I added 1 to the end of the filename, so within "results/Merge/" I had test1, test2, and test3. Each of these files has how long it took for a sort of 32, 64, 128, etc items. The output I dealt with looked like this:

```
*******************************
Insertion Sort of 16384 numbers:
_____________________________________
Started sort at  :  14:16:48:518   ||
Finished sort at :  14:16:49:187   ||
_____________________________________
comparisons:  67169362
```

The above is what the output of **java Insertion data/random/16384 16384** looked like. This, quite obviously, ran insertion sort on 16384 randomly-sorted items. It was appended to the file "results/Insertion/random/test2"

Merge Sort

Random Data			
n	**Expected counts: n(log(n))**	**Counts counted**	**Running Time**
32	160	107	00:00:375
64	384	277	00:00:041
128	896	668	00:00:027
256	2048	1602	00:00:017
512	4608	3721	00:00:018
1024	10240	8427	00:00:020
2048	22528	18908	00:00:027
4096	49152	41917	00:00:028
8192	106496	92036	00:00:047
16384	229376	200422	00:00:043
32768	491520	433669	00:00:079
65536	1048576	932891	00:00:107
131072	2228224	1996777	00:00:.156
262144	4718592	4256244	00:00:199
524288	9961472	9036968	00:00:750
1048576	20971520	19121071	00:00:908

Sorted Data			
n	**Expected counts (n/2)(log(n / 2))**	**Counts counted**	**Running Time**
32	64	64	00:00:023
64	160	160	00:00:017
128	384	384	00:00:017
256	896	896	00:00:018
512	2048	2048	00:00:017
1024	4608	4608	00:00:021
2048	10240	10240	00:00:024
4096	22528	22528	00:00:026
8192	49152	49152	00:00:035
16384	106496	106496	00:00:086
32768	229376	229376	00:00:054
65536	491520	491520	00:00:099
131072	1048576	1048576	00:00:140
262144	2228224	2228224	00:00:153
524288	4718592	4718592	00:00:326
1048576	9961472	9961472	00:00:714

Merge Sort (cont…)

	Reverse Data		
n	Expected counts (n/2)(log(n / 2))	Counts counted	Running Time
32	64	64	00:00:021
64	160	160	00:00:017
128	384	384	00:00:018
256	896	896	00:00:017
512	2048	2048	00:00:019
1024	4608	4608	00:00:025
2048	10240	10240	00:00:028
4096	22528	22528	00:00:043
8192	49152	49152	00:00:053
16384	106496	106496	00:00:079
32768	229376	229376	00:00:077
65536	491520	491520	00:00:071
131072	1048576	1048576	00:00:117
262144	2228224	2228224	00:00:144
24288	4718592	4718592	00:00:292
1048576	9961472	9961472	00:00:630

Merge sort has proved to be the best overall sort. It is efficient both in best and worst cases, and never took more than one second to sort any of the sets of data. Unlike Quicksort, it does not choke if the data is already sorted. All cases are theoretically of order n log n and when you look at the graph of n log n compared to the times it presents the same curve. I also found that generally merge sort took exactly as many as the theoretical number of comparisons required to complete the sort. I would be completely confident in suggesting someone use merge sort for nearly any application.

Quicksort*

Random Data			
n	**Expected counts: 1.38n(log(n))**	**Counts counted**	**Running Time**
32	221	265	00:00:195
64	530	588	00:00:020
128	1236	1517	00:00:017
256	2826	3371	00:00:019
512	6359	7746	00:00:019
1024	14131	17463	00:00:020
2048	31088	37743	00:00:023
4096	67830	87659	00:00:030
8192	146964	179286	00:00:085
16384	316538	405631	00:00:085
32768	678298	525013	00:00:123
65536	1447034	1855330	00:00:185
131072	3074949	3847470	00:00:248
262144	6511656	8492175	00:00:184
524288	13746831	17378283	00:00:254
1048576	28940697	36501968	00:00:514

Sorted Data			
n	**Expected counts: ((n-1)(n+2)/2)**	**Counts counted**	**Running Time**
32	527	527	00:00:016
64	2079	2079	00:00:017
128	8255	8255	00:00:017
256	32895	32895	00:00:019
512	131327	131327	00:00:026
1024	524799	524799	00:00:056
2048	2098175	2098175	00:00:209
4096	8390652	8390655	00:00:216
8192	335588527	33558527	00:00:343
16384	134225919	134225919	00:00:017
32768	536887295	536887295	00:04:405
65536	2147516415	2147516415	00:13:427
131072	8590000127	8590000127	00:58:784
262144	34360131585	34359869439	03:48:260
524288	137439739905	137439215615	15:02:638
1048576	549757386753	549756338175	~an hour

Quicksort* (cont…)

Reverse-sorted Data			
n	Expected counts: ((n-1)(n+2)/2) - 3	Counts counted	Running Time
32	524	559	00:00:343
64	2076	2143	00:00:019
128	8252	8383	00:00:017
256	32892	33151	00:00:019
512	131324	131839	00:00:027
1024	524796	525823	00:00:058
2048	2098172	2098423	00:00:339
4096	8390652	8392577	00:00:127
8192	335588524	33563255	00:00:290
16384	134225916	134159603	00:01:188
32768	536887292	536100299	00:03:748
65536	2147516412	2139964569	00:18:133
131072	8590000124	8531191541	00:59:000
262144	34360131582	33909130569	04:00:943
524288	137439739902	133939125867	21:00:246
1048576	549757386750	-----------------------	

*All the quicksorts were only performed twice due to time constraints.

Quicksort is excellent provided the data is not already in any sort of order, in which case it is just as bad as insertion sort normally is. Usually, quicksort runs only 38% slower than the best case of merge sort. If the right data is given quicksort can actually be faster than mergesort, but they are very close anyway. Quicksort does not seem to be stable or very predictable like merge sort is. Its time for completion greatly varies depending on the data given to it.

Insertion Sort

Random Data *

n	Expected counts: $n^2/4$	Counts counted	Running Time
32	256	281	00:00:064
64	1024	1089	00:00:020
128	4096	3854	00:00:019
256	16384	167382	00:00:022
512	65536	67437	00:00:027
1024	262144	266182	00:00:057
2048	1048576	1048616	00:00:161
4096	4194304	2110986	00:00:171
8192	16777216	16800231	00:00:275
16384	67108864	67191488	00:00:807
32768	268435456	269014025	00:03:313
65536	1073741824	1075709854	00:12:608
131072	4294967296	4297983951	00:55:.480
262144	17179869184	17169836669	03:36:751
524288	68719476736	68629574269	12:50:233
1048576	274877906944	275102402993 **	50:21:052

* -- These tests were only averaged over 2 tests because the first one was inadvertently removed
** -- This was only performed once due to the amount of time required

Sorted

n	Expected counts: n - 1	Counts counted	Running Time
32	31	31	00:00:200
64	63	63	00:00:042
128	127	127	00:00:018
256	255	255	00:00:018
512	511	511	00:00:023
1024	1023	1023	00:00:016
2048	2047	2047	00:00:023
4096	4095	4095	00:00:049
8192	8191	8191	00:00:032
16384	16383	16383	00:00:018
32768	32767	32767	00:00:021
65536	65535	65535	00:00:022
131072	131071	131071	00:00:032
262144	262143	262143	00:00:022
524288	524287	524287	00:00:035
1048576	1048575	1048575	00:00:035

Insertion Sort (cont…)

Reverse Sorted Data			
n	**Expected counts: (n-1)n/2**	**Counts counted**	**Running Time**
32	496	527	00:00:020
64	2016	2079	00:00:029
128	8128	8255	00:00:027
256	32640	32895	00:00:044
512	130816	131327	00:00:096
1024	523776	524799	00:00:110
2048	2096128	2098175	00:00:134
4096	8386560	8390653	00:00:206
8192	33550336	33558523	00:00:392
16384	134209536	134225893	00:01:265
32768	536854528	536887190	00:04:840
65536	2147450880	2147515964	00:16:322
131072	8589869056	8589998422	01:17:928
262144	34359607296	34359862531	05:03:863
524288	13759152128	137439188127	21:08:052
1048576	549755289600	------------------	~ 1 hour 20 minutes

Insertion sort, while easy to program, is abysmal when it comes to efficiency. Its best case is linear, which is excellent, but not very practical in most situations. Why sort something that is already sorted? Its worst case and average case are order n^2 which is nothing close to the n log n merge sort enjoys. It seems insertion sort is only useful up through about 4000 elements, after which one would be much better off to use one of the other sorts. One of the few pluses of insertion sort is that it is easy to get working correctly. It gave me the least trouble of all the methods.