MP3 Test Plan. Casey Hird.

Sort Type Performances

In this section test cases are defined for each of the 5 sorting functions implemented in MP3. These test cases are given in ‘test_cases.sh’. These test cases begin with one example of each sorting algorithm run with valgrind. The output from each of these cases ***IS SHOWN BELOW***. The second part of this testing evaluates each function at different combinations of input size, input sort direction, and output sort direction. The outputs from all these test cases are shown in ‘test_output.txt’ and the results for those cases in which the list was sorted to ascending order are summarized in the tables and on the graphs below. Each function is tested with randomly ordered input, then with ascending input, and finally with descending input. For each input case, 5 input sizes are tested, and these sizes are chosen to give a wide range of runtime values, the smallest being at most around a hundred milliseconds and the largest with a runtime greater than 1 second. Since there is variation in runtime, each of these cases is tested 3 times, and the average of these cases is shown as well. Finally, for each test case, a graph is included showing the relationship between the size of the list being sorted and the average runtime of the sorting function.

1. Bubble Sort

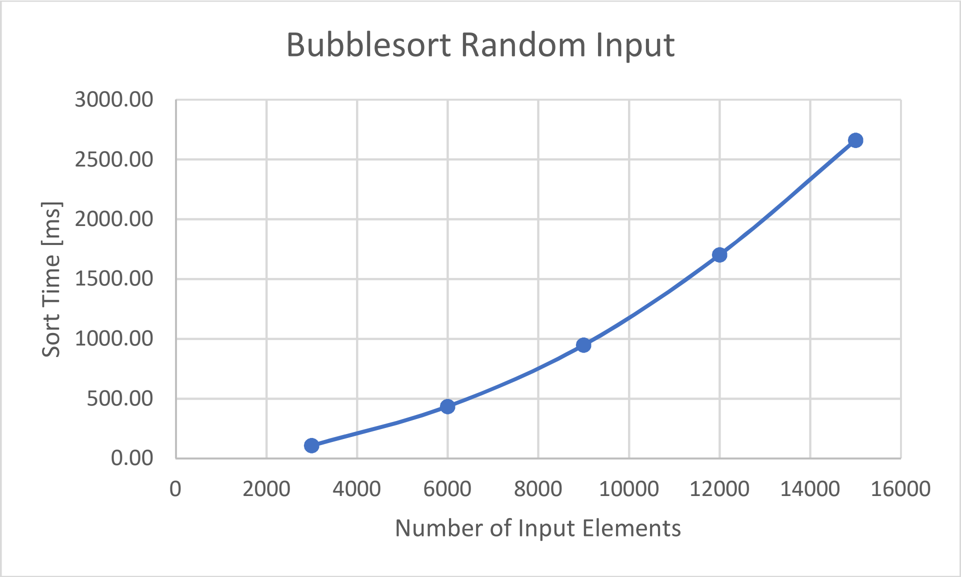
Bubble Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average RunTime [ms]
Random	3000	105.49	112.09	104.56	107.38
Random	6000	417.65	468.01	418.03	434.56
Random	9000	952.97	945.92	946.29	948.39
Random	12000	1708.43	1709.69	1691.21	1703.11
Random	15000	2659.82	2659.36	2665.63	2661.60
Ascending	3000	79.04	80.04	81.75	80.28
Ascending	6000	310.30	314.04	310.16	311.50
Ascending	9000	696.97	702.86	698.46	699.43
Ascending	12000	1328.45	1293.06	1245.31	1288.94
Ascending	15000	1916.36	1925.36	1932.61	1924.78
Descending	3000	102.01	102.39	103.85	102.75
Descending	6000	409.92	402.12	396.81	402.95
Descending	9000	889.90	898.07	892.89	893.62
Descending	12000	1586.43	1707.88	1607.08	1633.80
Descending	15000	2540.02	2476.03	2484.04	2500.03

Valgrind Output:

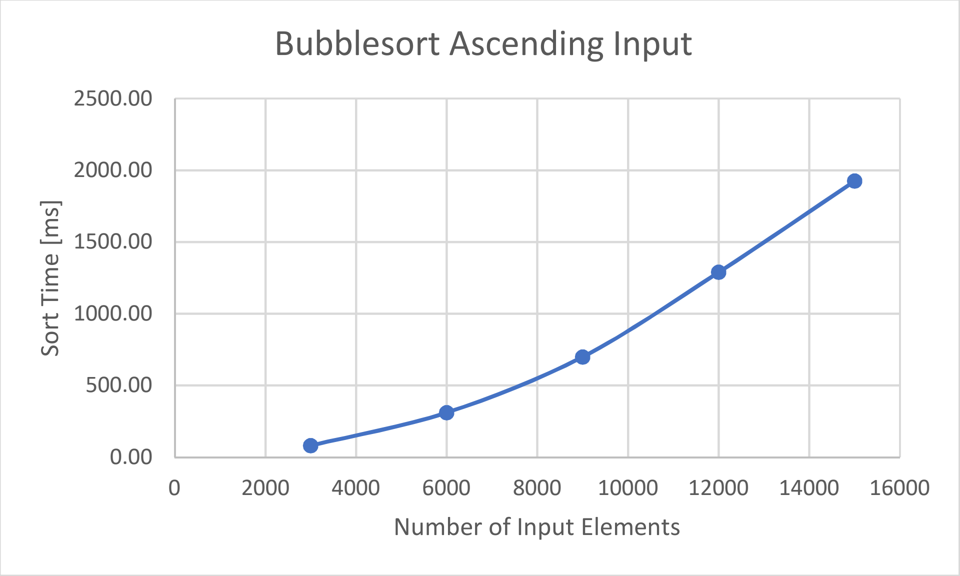
```
==15442== Memcheck, a memory error detector

==15442== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==15442== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==15442== Command: ./lab3 1
```

```
==15442== Parent PID: 15440

==15442==

==15442==

==15442== HEAP SUMMARY:

==15442==     in use at exit: 0 bytes in 0 blocks

==15442==   total heap usage: 45 allocs, 45 frees, 6,680 bytes allocated

==15442==

==15442== All heap blocks were freed -- no leaks are possible

==15442==

==15442== For counts of detected and suppressed errors, rerun with: -v

==15442== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

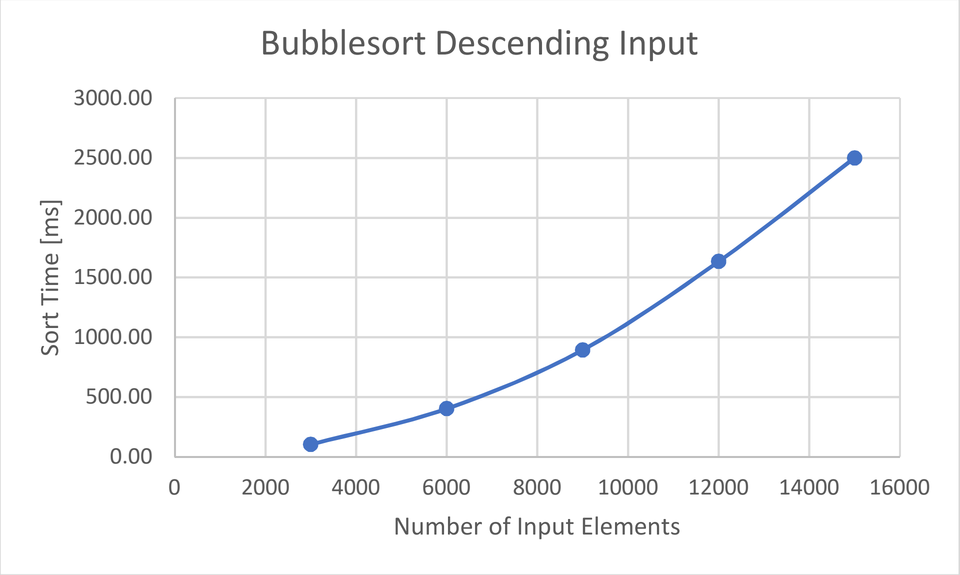
Sorting random input:



Sorting ascending input:



Sorting descending Input:



2. Insertion Sort

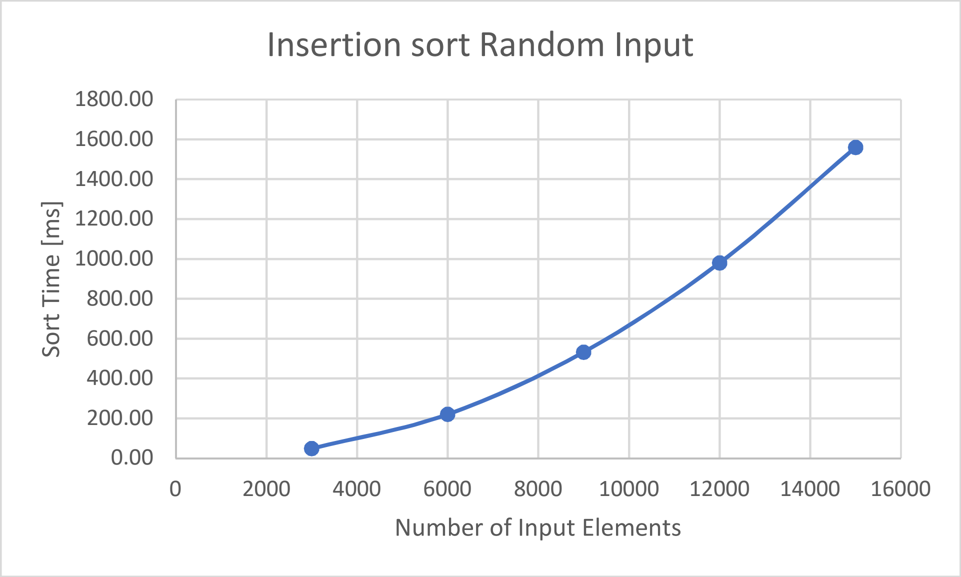
Insertion Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	3000	47.75	46.94	49.33	48.01
Random	6000	224.27	216.59	219.13	220.00
Random	9000	532.91	520.81	543.34	532.35
Random	12000	989.99	982.85	968.45	980.43
Random	15000	1567.49	1567.98	1544.58	1560.02
Ascending	3000	69.67	70.52	71.12	70.44
Ascending	6000	277.28	272.72	280.84	276.95
Ascending	9000	611.88	634.43	616.53	620.95
Ascending	12000	1094.03	1091.89	1102.77	1096.23
Ascending	15000	1721.76	1713.93	1695.52	1710.40
Descending	3000	0.33	0.28	0.31	0.31
Descending	6000	0.73	0.56	0.68	0.66
Descending	9000	1.00	1.05	0.89	0.98
Descending	12000	1.34	1.25	1.10	1.23
Descending	15000	1.70	1.64	1.47	1.60

Valgrind Output:

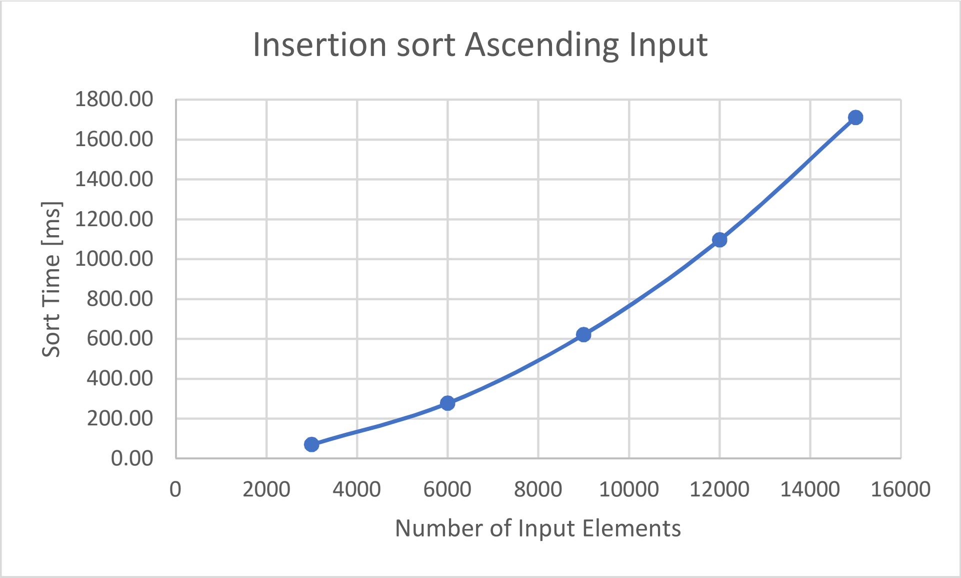
```
==15444== Memcheck, a memory error detector

==15444== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==15444== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==15444== Command: ./lab3 1

==15444== Parent PID: 15440

==15444==

==15444==

==15444== HEAP SUMMARY:

==15444==     in use at exit: 0 bytes in 0 blocks

==15444==   total heap usage: 66 allocs, 66 frees, 7,200 bytes allocated

==15444==

==15444== All heap blocks were freed -- no leaks are possible

==15444==

==15444== For counts of detected and suppressed errors, rerun with: -v
```

```
==15444== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

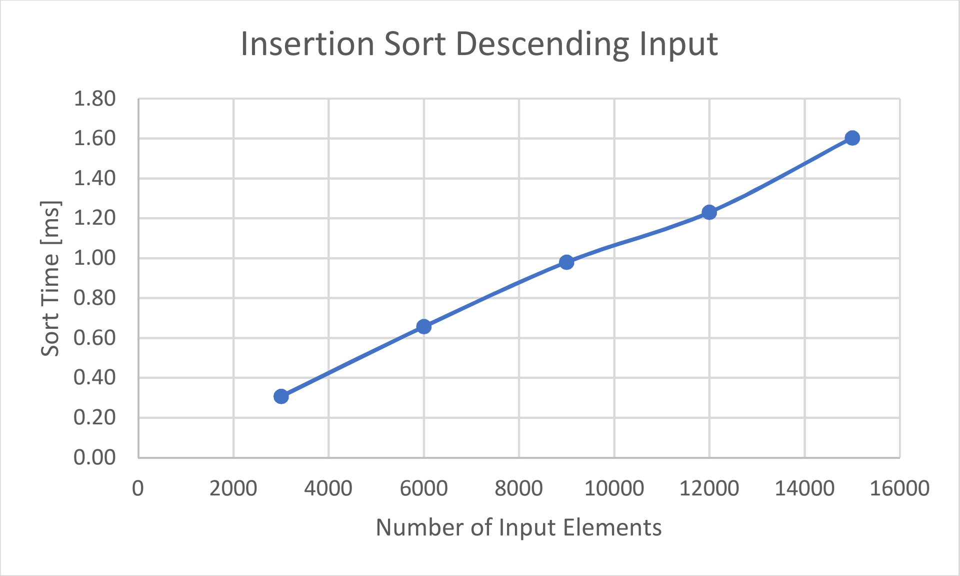
Sorting random input:



Sorting Ascending Input:



Sorting descending input:



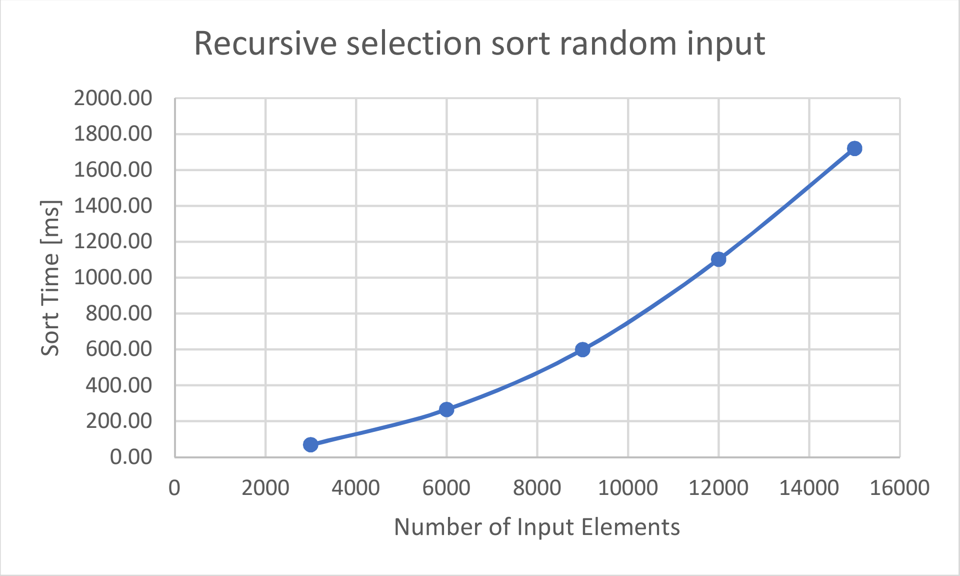
3. Recursive Selection Sort

Recursive Selection Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	3000	69.19	67.67	68.33	68.40
Random	6000	264.70	263.91	267.21	265.27
Random	9000	599.09	596.56	600.79	598.81
Random	12000	1063.83	1142.28	1099.81	1101.97
Random	15000	1660.33	1831.09	1667.61	1719.68
Ascending	3000	70.67	68.15	69.18	69.33
Ascending	6000	271.45	278.16	272.91	274.17
Ascending	9000	608.71	615.51	615.77	613.33
Ascending	12000	1080.19	1102.72	1090.28	1091.06
Ascending	15000	1786.89	1701.39	1749.97	1746.08
Descending	3000	72.72	69.80	70.84	71.12
Descending	6000	272.74	271.76	275.54	273.35
Descending	9000	616.04	609.15	613.95	613.05
Descending	12000	1085.49	1122.63	1099.44	1102.52
Descending	15000	1670.02	1691.51	1692.97	1684.83

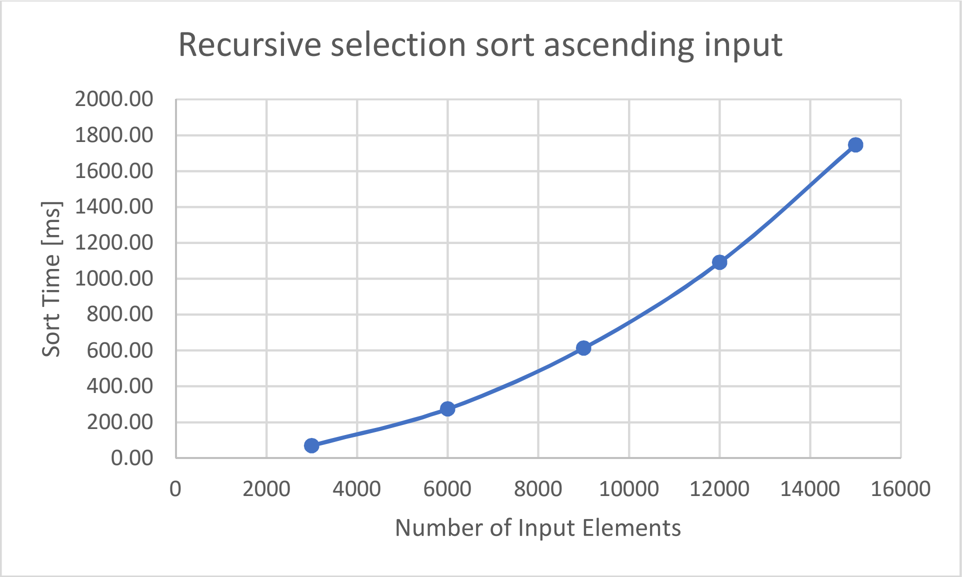
Valgrind Output:

```
==15446== Memcheck, a memory error detector
==15446== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==15446== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==15446== Command: ./lab3 1
==15446== Parent PID: 15440
==15446==
==15446==
==15446== HEAP SUMMARY:
==15446==     in use at exit: 0 bytes in 0 blocks
==15446==   total heap usage: 45 allocs, 45 frees, 6,680 bytes allocated
==15446==
==15446== All heap blocks were freed -- no leaks are possible
==15446==
==15446== For counts of detected and suppressed errors, rerun with: -v
==15446== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

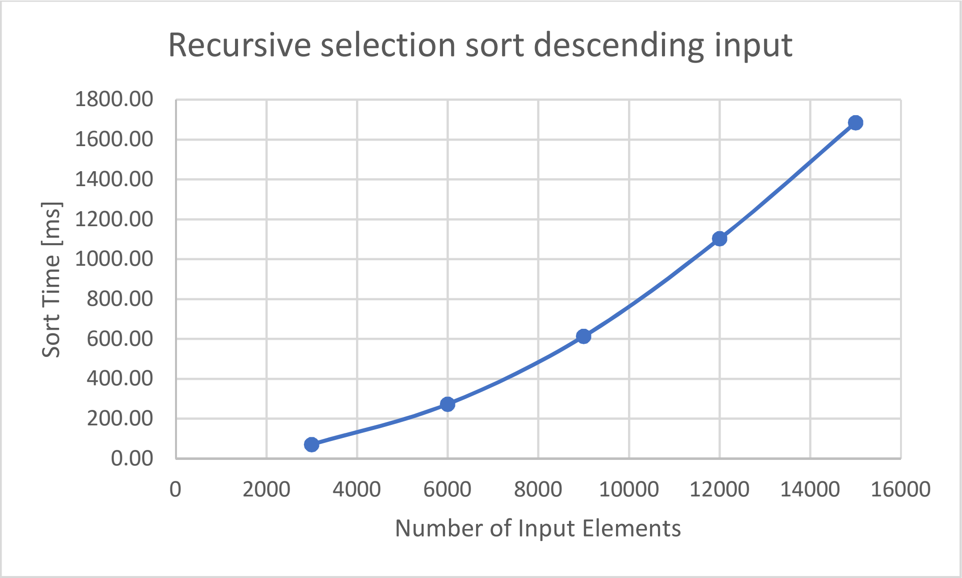
Sorting random input:



Sorting ascending input:



Sorting descending input:



4. Iterative Selection Sort

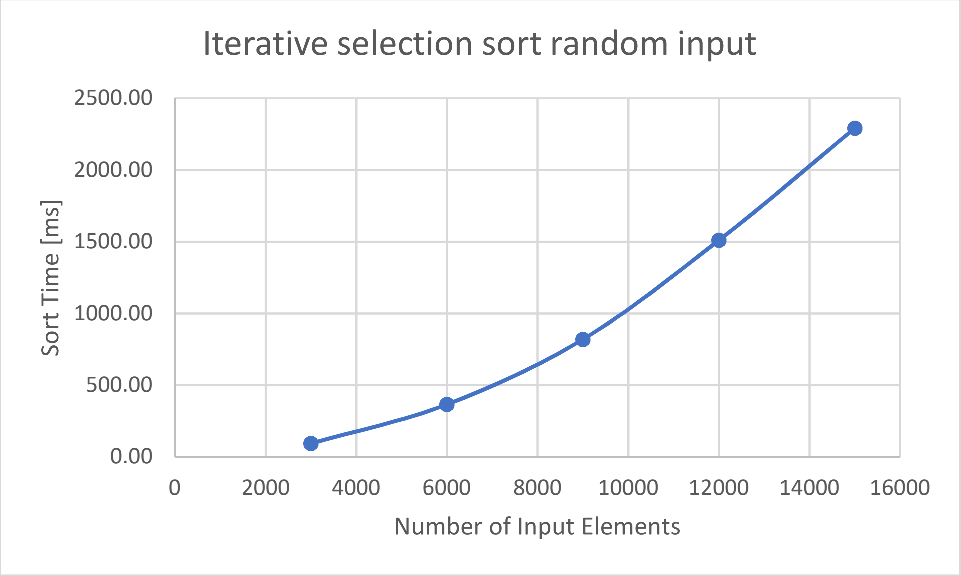
Iterative Selection Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	3000	94.39	92.66	92.30	93.12
Random	6000	368.75	362.56	367.15	366.15
Random	9000	820.87	819.24	819.24	819.78
Random	12000	1465.32	1608.57	1455.66	1509.85
Random	15000	2299.69	2283.74	2288.80	2290.74
Ascending	3000	94.14	97.06	96.64	95.95
Ascending	6000	369.79	370.25	367.85	369.30
Ascending	9000	829.45	831.10	829.17	829.91
Ascending	12000	1581.97	1477.78	1469.31	1509.69
Ascending	15000	2292.80	2317.88	2302.39	2304.36
Descending	3000	94.23	94.60	95.42	94.75
Descending	6000	371.92	368.05	383.24	374.40
Descending	9000	825.25	854.50	822.71	834.15
Descending	12000	1470.32	1472.45	1470.96	1471.24
Descending	15000	2346.06	2292.93	2298.71	2312.57

Valgrind Output:

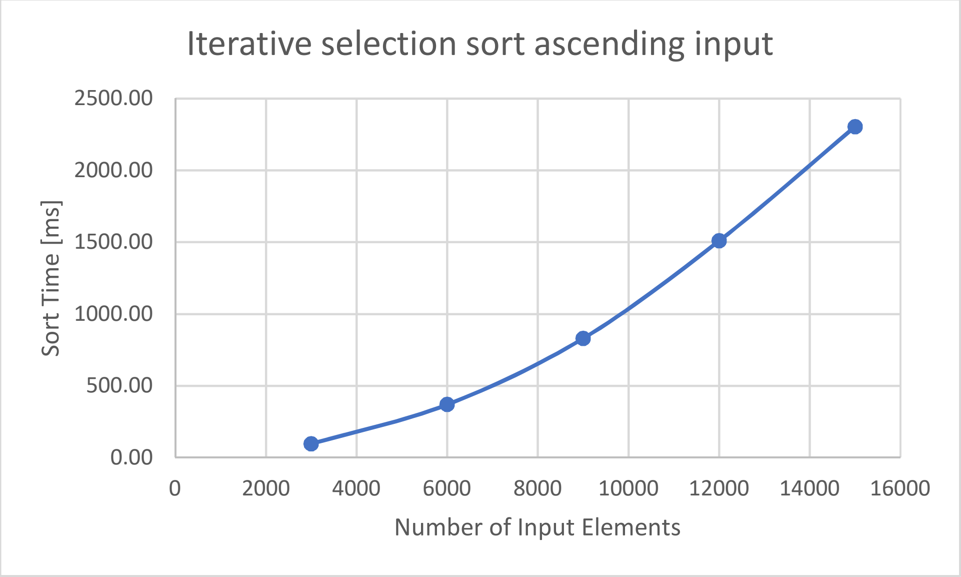
```
==15448== Memcheck, a memory error detector

==15448== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.

==15448== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info

==15448== Command: ./lab3 1

==15448== Parent PID: 15440

==15448==

==15448==

==15448== HEAP SUMMARY:

==15448==     in use at exit: 0 bytes in 0 blocks

==15448==   total heap usage: 45 allocs, 45 frees, 6,680 bytes allocated

==15448==

==15448== All heap blocks were freed -- no leaks are possible

==15448==

==15448== For counts of detected and suppressed errors, rerun with: -v
```

```
==15448== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

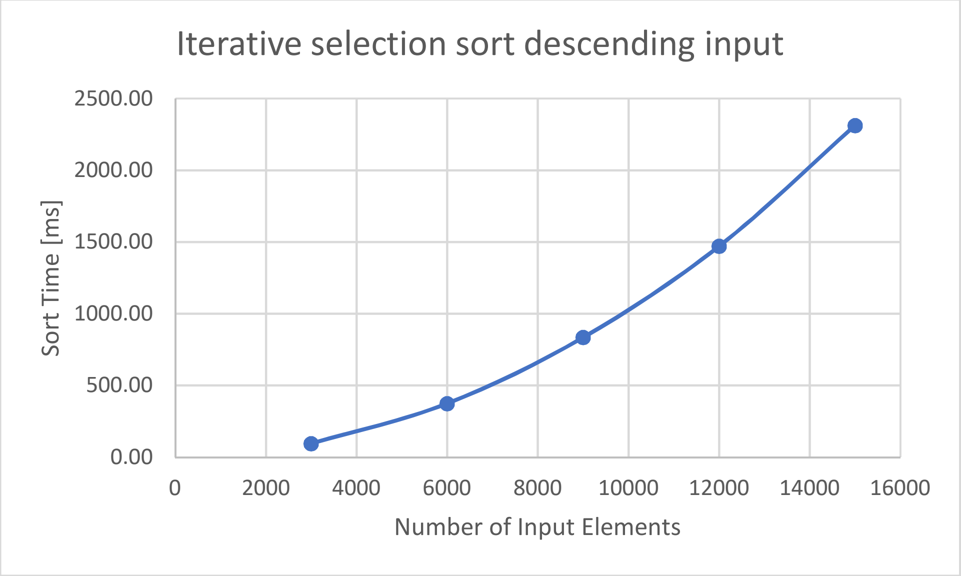
Sorting random input:



Sorting ascending input:



Sorting descending input:



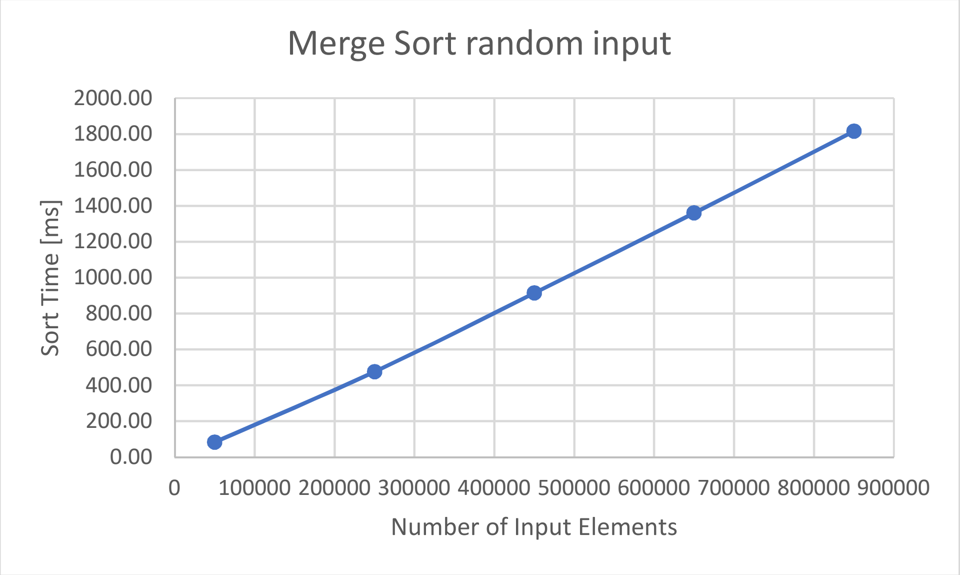
5. Merge Sort

Merge Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	50000	83.37	83.92	83.58	83.62
Random	250000	471.49	481.15	475.82	476.15
Random	450000	906.00	908.78	928.05	914.28
Random	650000	1349.82	1350.18	1379.57	1359.86
Random	850000	1821.87	1814.68	1811.12	1815.89
Ascending	50000	74.51	73.02	73.12	73.55
Ascending	250000	401.62	413.65	401.67	405.65
Ascending	450000	759.23	761.53	758.11	759.62
Ascending	650000	1131.25	1126.00	1128.88	1128.71
Ascending	850000	1488.00	1536.14	1509.29	1511.14
Descending	50000	75.18	73.88	84.80	77.95
Descending	250000	410.81	423.76	438.39	424.32
Descending	450000	763.04	776.95	785.18	775.06
Descending	650000	1130.90	1136.06	1147.42	1138.13
Descending	850000	1519.93	1502.13	1528.39	1516.82

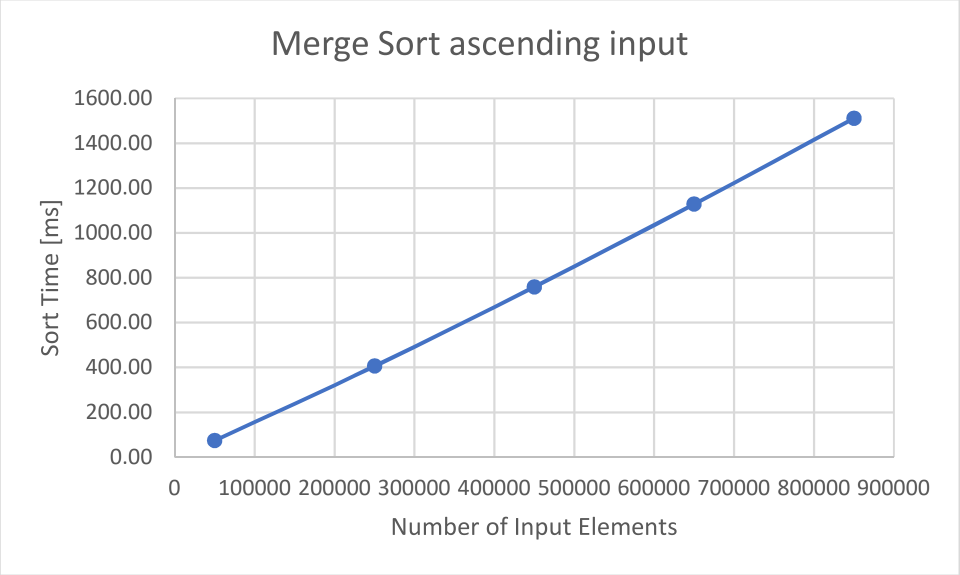
Valgrind Output:

```
==15450== Memcheck, a memory error detector
==15450== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==15450== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==15450== Command: ./lab3 1
==15450== Parent PID: 15440
==15450==
==15450==
==15450== HEAP SUMMARY:
==15450==     in use at exit: 0 bytes in 0 blocks
==15450==   total heap usage: 190 allocs, 190 frees, 11,072 bytes allocated
==15450==
==15450== All heap blocks were freed -- no leaks are possible
==15450==
==15450== For counts of detected and suppressed errors, rerun with: -v
==15450== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

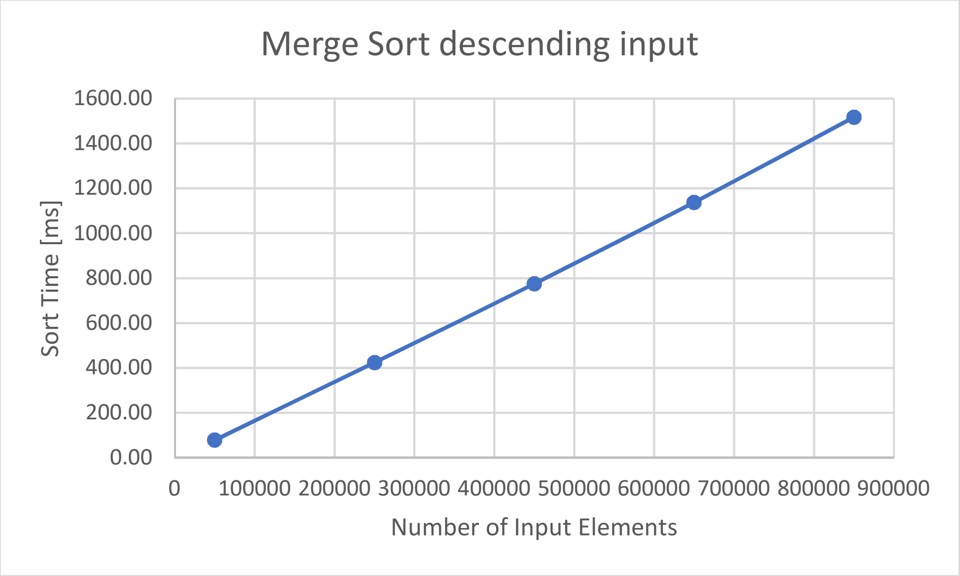
Sorting random input:



Sorting ascending input:



Sorting descending input:



Sort Function Comparisons

1. For lists that are initially random, we see very similar runtimes for the first 4 functions. Each of these has around 100 millisecond run time to sort a list of 3000 elements, and around 1.5-2 second runtime to sort 15000 elements. We can note from this that the recursive and iterative implementations of the selection sort algorithm have very similar run times. This is what we would expect, because the implementation of the algorithm will affect the constant times required to perform certain operations, but the time complexity of the algorithm will not be changed. This is also the reason we see similar runtimes for the first 4 functions. The first function implements a bubble sort algorithm, the second an insertion sort algorithm, and the third and fourth a selection sort algorithm. Each of these has worst case run time complexity of O(n^2), so we would expect that at large values of n, like those we have tested, the run times of all these functions should be very similar. The merge sort function is very different, showing a drastic improvement in run time. The merge sort function is several orders of magnitude faster than the other functions—according to the data above, merge sort could sort around 600,000 elements in 1 second, while the other 4 functions could sort around 12000 elements in 1 second. This disparity is to be expected though, because the complexity of the merge sort algorithm is O(nlogn), which is, in the worst case, much faster than the other functions which have O(n^2) complexity. This means that we expect merge sort to be able to sort much larger lists than the other functions with similar run times, and we see that this is the case.
2. We see a slight improvement in performance in our bubble sort algorithm based on the sorting of the input list. If the input list is in ascending order and we are sorting to ascending order, then the function never needs to swap any elements. The decrease in run time is not excessive, because the function must still perform all the comparisons between elements, but there is a significant change since there is never any reason to swap elements. The real significant disparity in runtime is seen when a sorted list is given as input to insertion sort. Specifically, the run time of insertion sort falls by several orders of magnitude when the input list is in descending order and we are sorting into ascending order. This occurs because when the input is already sorted in reverse order, we only need to make a single to insert each element into the new list. So, where for a random input list, inserting the nth node might take n-1 comparisons to each of the nodes inserted previously, if the list is already sorted the in the opposite order, then the node currently at the head will be already less than the node we are currently at the head of the new list. This means that in this case we are simply inserting each new node at the head of the new list, so each insertion requires 1 comparison, as opposed to n-1 comparisons. We do not see any disparity in either of the other two functions. In selections sort, we still must sort the entire list to find the minimum or maximum value for each iteration, regardless of whether the input list was sorted. For merge sort, we still must make all of the recursive calls and then merge the lists, so there will be no difference if the input list is sorted.

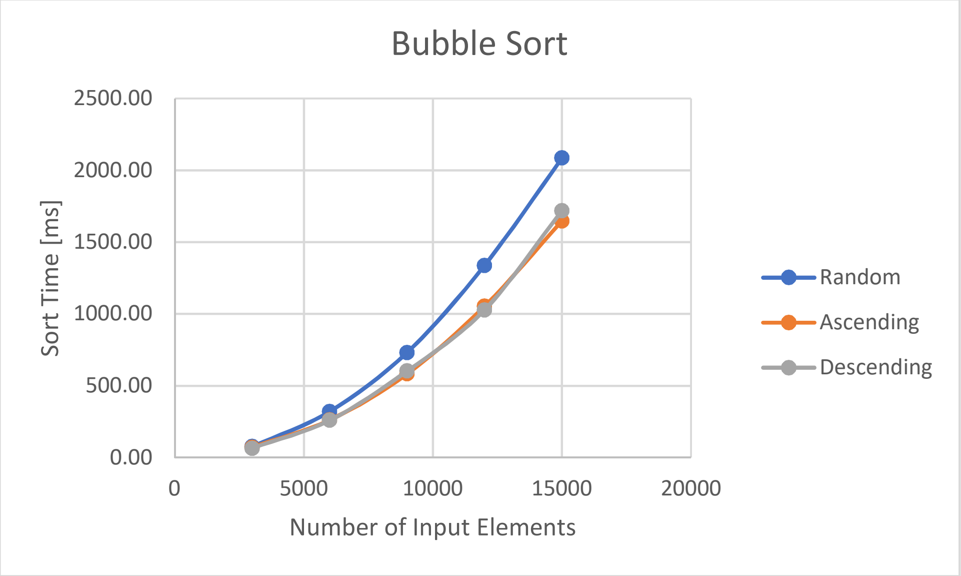
Bonus

Sort Type Performances

Here we perform the same analysis as above, the only difference being that we compiled the files in MP3 without the flags -Wall and -g and with the -O flag to have the compiler optimize our code. The results of this are shown below.

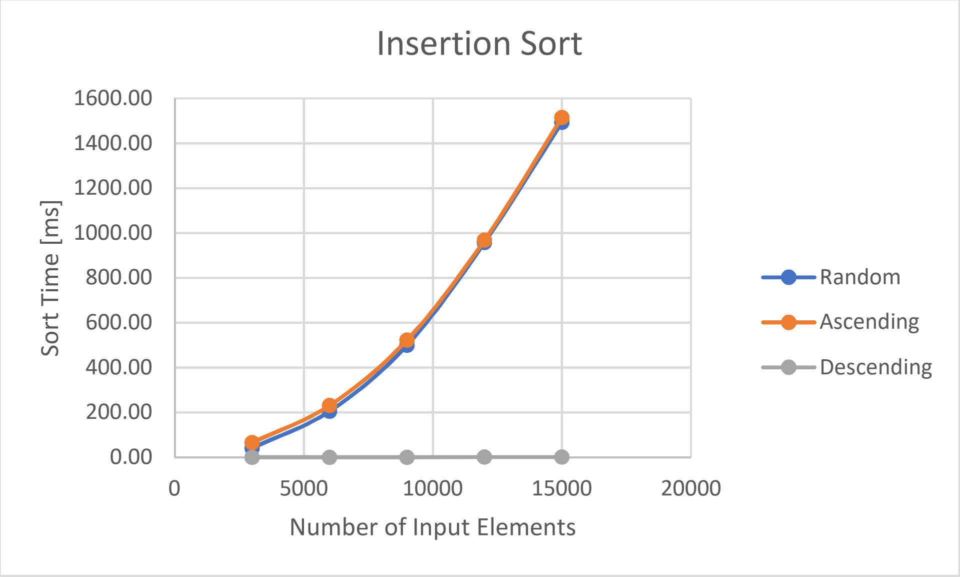
1. Bubble Sort

Bubble Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average RunTime [ms]
Random	3000	75.89	76.48	76.69	76.35
Random	6000	326.55	317.60	318.50	320.88
Random	9000	729.99	733.43	732.74	732.05
Random	12000	1304.53	1332.94	1377.84	1338.44
Random	15000	2139.97	2057.48	2061.35	2086.27
Ascending	3000	65.61	83.28	65.67	71.52
Ascending	6000	264.19	260.34	261.13	261.89
Ascending	9000	574.97	582.58	592.42	583.32
Ascending	12000	1039.29	1107.30	1015.86	1054.15
Ascending	15000	1645.63	1644.69	1655.62	1648.65
Descending	3000	65.66	66.03	63.74	65.14
Descending	6000	256.30	262.69	258.86	259.28
Descending	9000	580.36	633.38	594.71	602.82
Descending	12000	1030.12	1031.38	1019.66	1027.05
Descending	15000	1770.91	1716.21	1672.05	1719.72



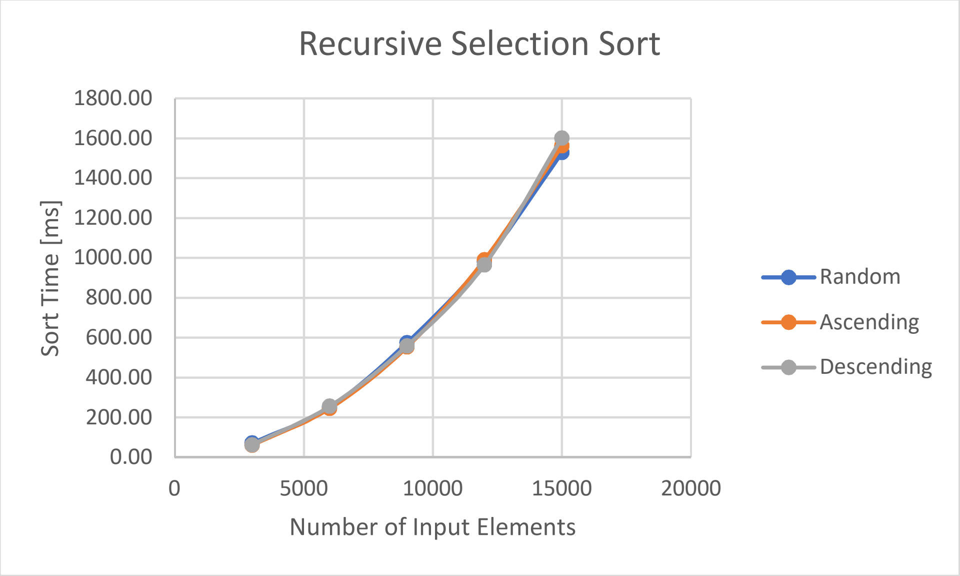
2. Insertion Sort

Insertion Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	3000	42.12	42.04	40.63	41.60
Random	6000	185.51	228.70	203.57	205.93
Random	9000	532.64	473.16	494.93	500.24
Random	12000	938.85	991.21	943.59	957.88
Random	15000	1464.37	1555.10	1462.54	1494.00
Ascending	3000	73.09	61.25	62.81	65.72
Ascending	6000	233.11	230.06	231.23	231.47
Ascending	9000	519.09	519.17	530.34	522.87
Ascending	12000	963.59	973.19	968.15	968.31
Ascending	15000	1558.72	1506.90	1477.83	1514.48
Descending	3000	0.28	0.23	0.27	0.26
Descending	6000	0.54	0.54	0.55	0.54
Descending	9000	0.81	0.81	0.83	0.82
Descending	12000	1.79	1.14	1.07	1.33
Descending	15000	1.37	1.30	1.46	1.38



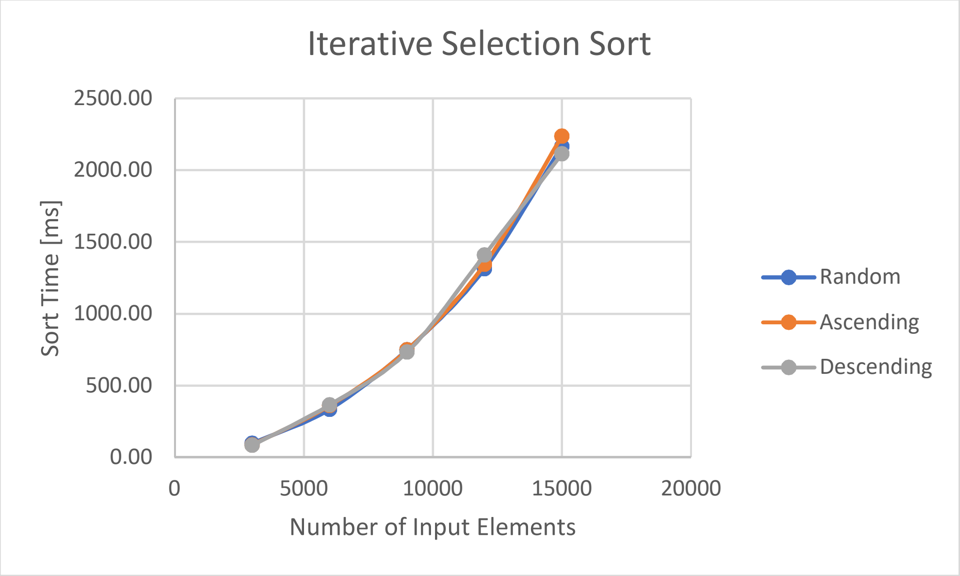
3. Recursive Selection Sort

Recursive Selection Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	3000	60.93	89.05	64.38	71.45
Random	6000	249.68	262.81	242.00	251.50
Random	9000	537.04	578.07	610.63	575.25
Random	12000	965.28	945.68	1034.62	981.86
Random	15000	1524.78	1498.99	1560.56	1528.11
Ascending	3000	60.35	61.88	62.81	61.68
Ascending	6000	241.17	239.62	256.11	245.63
Ascending	9000	582.56	543.69	535.47	553.91
Ascending	12000	947.72	997.46	1028.80	991.33
Ascending	15000	1497.56	1572.00	1619.74	1563.10
Descending	3000	62.62	61.78	64.05	62.82
Descending	6000	258.80	256.76	253.67	256.41
Descending	9000	560.75	535.60	582.01	559.45
Descending	12000	955.74	987.25	952.27	965.09
Descending	15000	1638.24	1601.23	1565.04	1601.50



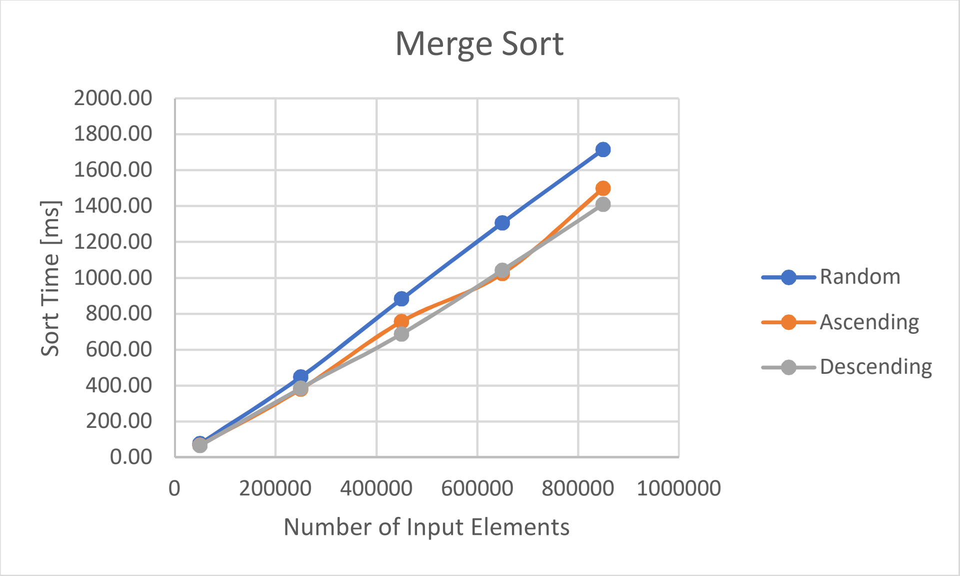
4. Iterative Selection Sort

Iterative Selection Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	3000	120.57	83.76	87.58	97.30
Random	6000	342.30	331.04	330.38	334.57
Random	9000	751.13	739.95	759.44	750.17
Random	12000	1335.85	1308.55	1295.98	1313.46
Random	15000	2077.84	2210.25	2208.23	2165.44
Ascending	3000	86.42	85.71	86.38	86.17
Ascending	6000	323.79	435.36	321.17	360.11
Ascending	9000	734.16	754.73	756.98	748.62
Ascending	12000	1366.99	1320.88	1345.74	1344.54
Ascending	15000	2178.23	2297.26	2234.43	2236.64
Descending	3000	80.65	83.70	86.52	83.62
Descending	6000	376.66	366.04	353.77	365.49
Descending	9000	723.16	726.69	751.98	733.94
Descending	12000	1406.45	1324.89	1497.28	1409.54
Descending	15000	2046.15	2074.29	2221.12	2113.85



5. Merge Sort

Merge Sort					
List Type	Size	Runtime 1 [ms]	Runtime 2 [ms]	Runtime 3 [ms]	Average Runtime [ms]
Random	50000	76.89	75.40	75.40	75.90
Random	250000	440.65	439.30	463.22	447.72
Random	450000	884.32	923.05	841.55	882.97
Random	650000	1342.82	1259.21	1317.03	1306.35
Random	850000	1798.82	1668.52	1678.24	1715.19
Ascending	50000	63.36	64.14	76.87	68.12
Ascending	250000	367.57	363.02	410.86	380.48
Ascending	450000	709.41	722.84	839.18	757.14
Ascending	650000	1008.17	1022.90	1041.89	1024.32
Ascending	850000	1617.64	1503.38	1372.52	1497.85
Descending	50000	65.20	65.77	64.81	65.26
Descending	250000	431.90	361.42	363.95	385.76
Descending	450000	694.59	685.46	683.05	687.70
Descending	650000	1070.36	1042.60	1012.12	1041.69
Descending	850000	1416.39	1445.61	1368.85	1410.28



Analysis

As expected, the run times for each function were reduced slightly by optimizing using the -O compiler flag. However, we can see that the complexities for each function remained the same. This is what we would expect, since the actual form of the functions is not changing, so the algorithms implemented by these functions does not change, so neither will their complexities. We can see this is true by analyzing the graphs shown above. Recall that the first function is bubble sort, the second is insertion sort, and the third and fourth are selection sort—all of which are algorithms with O(n^2) complexity. This is supported by the graphs shown because we see that the run time of these functions increases with approximately an n^2 relationship to the size of the list, n. Similarly, we see that the merge sort function holds to its expected complexity O(nlogn). In conclusion, we see the expected results—compiler optimization helps to reduce the constant run times of these functions, reducing the run time for any given list size, but the complexity of each function remains the same.