INSERTION SORT

Primitive :

- For m = 10, n = 10000 and d = -1:

  Timer (generate): 1.5ms real (average of 10 test cases)
  Timer (sort): 1011.8ms real (average of 10 test cases)

- For m = 10, n = 25000 and d = -1:

  Timer (generate): 3.2ms real (average of 10 test cases)
  Timer (sort): 11,907.4ms real (average of 10 test cases)

- For m = 10, n = 50000 and d = -1:

  Timer (generate): 6.7 ms real (average of 10 test cases)
  Timer (sort): 22,803.5ms real (average of 10 test cases)

- For m = 10, n = 100000 and d = -1:

  Timer (generate): 13.9 ms real (average of 10 test cases)
  Timer (sort): 90,464ms real (average of 10 test cases)

- For m = 10, n = 250000 and d = -1:

  Timer (generate): (41 + 30 + 30 + 38 + 39 + 39 + 40 + 21 + 38 + 35)/10 = 35.1 ms real ; (average of 10 test cases)
  Timer (sort): (543548 + 547537 + 550653 + 537017 + 544089 + 533805 + 542442 + 534824 + 553559 + 537688)/10 = 542,516.2ms real(average of 10 test cases)

  The above test-case takes approximately 9.06 minutes to perform insertion sort through the generated data. And since this is a primitive insertion sort algorithm with time complexity of O(n^3) it will take longer for higher value inputs of n.

- For m = 10, n = 500000 and d = -1:

  This will take more than 10 minutes to run the insertion sort algorithm , hence stopping here. The above test case takes longer as the size of n has increased to 500000 , showing that this insertion sort code is not the most efficient and if improved needs to be able to handle bigger inputs like the one above

---

- For m = 10, n = 10000 and d = 0:

  Timer (generate): 3ms real (average of 10 test cases)
  Timer (sort): 708.2ms real (average of 10 test cases)

- For m = 10, n = 25000 and d = 0:

  Timer (generate): 7.6ms real (average of 10 test cases)
  Timer (sort): 4,634ms real (average of 10 test cases)

- For m = 10, n = 50000 and d = 0:

  Timer (generate): 15.8ms real (average of 10 test cases)
  Timer (sort): 19,246.6ms real (average of 10 test cases)

- For m = 10, n = 100000 and d = 0:

  Timer (generate): 30.3ms real (average of 10 test cases)
  Timer (sort): 80,457.9ms real (average of 10 test cases)

- For m = 10, n = 250000 and d = 0:

  Timer (generate): 75ms real (average of 10 test cases)
  Timer (sort): 605,098ms real ;(average of 10 test cases)

  This takes a little bit more than 10 minutes to finish , hence we're stopping here.
  This is the case where a randomly sorted array is passed into the naive insertion sort algorithm. We can see that it takes a fair bit of time to run.

- For m = 10, n = 10000 and d = 1:

  Timer (generate): 2.2ms real (average of 10 test cases)
  Timer (sort): 0ms real (average of 10 test cases)

- For m = 10, n = 25000 and d = 1:

  Timer (generate): 3.6ms real (average of 10 test cases)
  Timer (sort): 1ms real (average of 10 test cases)

- For m = 10, n = 50000 and d = 1:

  Timer (generate): 8.3ms real (average of 10 test cases)
  Timer (sort): 2ms real (average of 10 test cases)

- For m = 10, n = 100000 and d = 1:

  Timer (generate): 16.6ms real (average of 10 test cases)
  Timer (sort): 3.4ms real (average of 10 test cases)

- For m = 10, n = 250000 and d = 1:

  Timer (generate): 35.8ms real (average of 10 test cases)
  Timer (sort): 7.5ms real (average of 10 test cases)

- For m = 10, n = 500000 and d = 1:

  Timer (generate): 62.9ms real (average of 10 test cases)
  Timer (sort): 13.2ms real (average of 10 test cases)

- For m = 10, n = 1000000 and d = 1:

  Timer (generate): 112ms real (average of 10 test cases)
  Timer (sort): 23.8ms real (average of 10 test cases)

- For m = 10, n = 2500000 and d = 1:

  Timer (generate): 265.3ms real (average of 10 test cases)
  Timer (sort): 58.6ms real (average of 10 test cases)

In the above findings for the value of d as 1 i.e sorted vector we see that the run times are shorter , this indicates that in the best case scenario (sorted input) the program works in reasonable time even for higher values on n.

But if the input is unsorted or reverse sorted , this program becomes less efficient , and takes a long time to run.

***Checking with different values of m* :**

- For m = 25, n = 10000 and d = -1:

  Timer (generate): 3.3 ms real (average of 10 test cases)
  Timer (sort): 2,246.6 ms real(average of 10 test cases)

- For m = 25, n = 25000 and d = -1:

  Timer (generate): 6 ms real (average of 10 test cases)
  Timer (sort): 16,058.4ms real(average of 10 test cases)

- For m = 25, n = 50000 and d = -1:

  Timer (generate): 13 ms real (average of 10 test cases)
  Timer (sort): 59,437.3ms real (average of 10 test cases)

- For m = 25, n = 100000 and d = -1:

  Timer (generate): 41ms real (average of 5 test cases)
  Timer (sort): 396,456.5ms real (average of 5 test cases)

  The average of 5 test cases is taken here as each test case takes about 7 mins to sort the reverse sorted array input.

  This takes approximately 7 minutes to run hence we realize that this algorithm is at its threshold. The time taken to process a big input size as seen in the previous test case increases in the order of O(n^3). We can improve the algorithm for it to fit the tightest bound and reduce the time complexity from O(n^3) -to O(n^2).

- For m = 25, n = 250000 and d = -1:

  This takes more than 10 mins to accomplish and hence we stop testing for values greater than 250000

- For m = 25, n = 10000 and d = 0:

  Timer (generate): 5.1 ms real (average of 10 test cases)
  Timer (sort): 1,647.2 ms real(average of 10 test cases)

- For m = 25, n = 25000 and d = 0:

Timer (generate): 13.6 ms real (average of 10 test cases)
Timer (sort): 10,357ms real (average of 10 test cases)

- For m = 25, n = 50000 and d = 0:

  Timer (generate): 33.25 ms real (average of 10 test cases)
  Timer (sort): 44,762.3ms real (average of 10 test cases)

- For m = 25, n = 100000 and d = 0:

  Timer (generate): 76ms real (average of 5 test cases)
  Timer (sort): 433,916ms real (average of 5 test cases)

  Since the test time for this takes approximately 7 to 8 mins we have taken 5 test cases and this lets us know that if value of n = 250000 or greater it takes longer than 10 mins and hence we stop simulations here.

- For m = 25, n = 250000 and d = 0:

  This takes more than 10 mins to accomplish and hence we stop testing for values greater than or equal to 250000

  In the above test cases we input a randomly sorted array into the naive insertion sort algorithm and find that the time taken to run the program increases in the order of O(n^2) but in this insertion sort algorithm increases in the order of O(n^3).

- For m = 25, n = 10000 and d = 1: In this case time taken is 1ms real (average of 10 test cases)

- For m = 25, n = 50000 and d = 1: In this case time taken is 3.7ms real (average of 10 test cases)

- For m = 25, n = 100000 and d = 1: In this case time taken is 9.62ms real (average of 10 test cases)

- For m = 25, n = 500000 and d = 1: In this case time taken is 38.5ms real (average of 10 test cases)

- For m = 25, n = 1000000 and d = 1: In this case time taken is 89.3ms real (average of 10 test cases)

- For m = 25, n = 2500000 and d = 1: In this case time taken is 312.8ms real (average of 10 test cases)

  When the values of d is 1 it corresponds to an input array which is sorted. This is the best case scenario for any insertion sort algorithm hence takes shorter time to run.

- For m = 50, n = 10000 and d = -1: In this case time taken is 8,016.2ms real (average of 10 test cases)

- For m = 50, n = 50000 and d = -1: In this case time taken is 38,668ms real (average of 10 test cases)

- For m = 50, n = 100000 and d = -1: In this it takes more than 10 mins to run the sorting algorithm for an reverse sorted array.. This is due to the fact that the time taken for larger values of n and vector m , increases the time taken to run the code.

- For m = 50, n = 10000 and d = 0: In this case time taken is 3,289.8ms real (average of 10 test cases)

- For m = 50, n = 50000 and d = 0: In this case time taken is 232,897.2ms real (average of 5 test cases).
  This already takes a long time to sort through the whole random array and here is where the ceiling for this algorithm is reached and for larger m&n values for insertion sort we need to improve the algorithm further

---

- For m = 50, n = 10000 and d = 1: In this case time taken is 1ms real (average of 10 test cases)

- For m = 50, n = 25000 and d = 1: In this case time taken is 3.2ms real (average of 10 test cases)
- For m = 50, n = 50000 and d = 1: In this case time taken is 8.9ms real (average of 10 test cases)
- For m = 50, n = 100000 and d = 1: In this case time taken is 24.4ms real (average of 10 test cases)
- For m = 50, n = 250000 and d = 1: In this case time taken is 56.6ms real (average of 10 test cases)
- For m = 50, n = 500000 and d = 1: In this case time taken is 115.75ms real (average of 10 test cases)
- For m = 50, n = 1000000 and d = 1: In this case time taken is 196ms real (average of 10 test cases)
- For m = 50, n = 2500000 and d = 1: In this case time taken is 375.5ms real (average of 10 test cases)

Sorted array , best case scenario and works for all cases given in the question.

IMPROVED INSERTION SORT ALGORITHM

For a reverse sorted array i.e d = -1 the best case scenario for the naive insertion sort algorithm where n = 10000 m = 10 d = -1 the time taken on average was 1011.8 ms whereas for the improved insertion sort algorithm the time taken is 102.7ms .. This is because the naive

insertion sort algorithm has many inner loops inside the main loop hence increasing time complexity . It doesn't make much of a difference with respect to time taken to run small sized inputs of n and m but as the input array size increases the time complexity comes into play and hence the naive insertion sort algorithm takes longer time to run.

Now testing for edge case values of n and m and d where the previous algorithm took longer than the specified 10 min threshold

- For m = 10, n = 10000 and d = -1: The average of 10 run times is 487.3 ms
- For m = 10, n = 50000 and d = -1: The average of 10 run times is 2,692.2 ms
- For m = 10, n = 100000 and d = -1: The average of 10 run times is 11,172.5 ms
- For m = 10, n = 250000 and d = -1: The average of 10 run times is 53,507ms
- For m = 10, n = 500000 and d = -1: The average of 10 run times is 235,375ms
- For m = 10, n = 1000000 and d = -1: The average of 10 run times is greater than 10 minutes hence we halt the simulations here.

Since the last test case has a run time which is greater than 10 mins we stop the simulation here . For bigger values of n merge sort is a better algorithm to use as we will see in the later sections of the report. Regardless of the efficiency of the new algorithm which is O(n^2) it is still not suited for large inputs of m and n. We will see more about this as we go through the remaining test cases.

But comparing this to the naive insertion sort algorithm , it's more efficient. If the input sizes are small for example n=5 m=5 and the input array is either a reverse sorted array, sorted array or a randomly generated array, then the run times of the naive and the improved insertion sort algorithm are the same. But as the input size increases the efficiency of the improved insertion sort algorithm becomes better than the naive insertion sort algorithm. For example - for the input m = 10 , n =50000 and d =-1, the naive insertion sort algorithm has a run time : 22,803.5ms whereas the improved insertion sort algorithm has a run time of 2,692.2 ms.

- For m = 10, n = 50000 and d = 0: The average of 10 run times is 2,392 ms
- For m = 10, n = 100000 and d = 0: The average of 10 run times is 4,161 ms
- For m = 10, n = 250000 and d = 0: The average of 10 run times is 19,897.8ms
- For m = 10, n = 500000 and d = 0: The average of 10 run times is 102,671.6ms

- For m = 10, n = 1000000 and d = 0: This takes about 409,507ms to run, but in real time takes more than 10 minutes and hence we are stopping the simulation here.

We have reached the threshold of time taken to let the algorithm run. Hence we stop here. What we've found here is that a randomly sorted array is an average case scenario whereas the reverse sorted arrays are worst case scenarios as reverse sorted arrays have larger runtimes as compared to randomly sorted arrays as we can see from the results above.

As we all know when the array is sorted i.e d = 1 , that is the best case scenario and hence there isn't much difference in the runtimes of the two algorithms used. And the following limited test cases show the same.

- For m = 10, n = 10000 and d = 1: The average of 10 run times is 0.3 ms
- For m = 10, n = 50000 and d = 1: The average of 10 run times is 1.3 ms
- For m = 10, n = 100000 and d = 1: The average of 10 run times is 5.7 ms
- For m = 10, n = 500000 and d = 1: The average of 10 run times is 14.8 ms
- For m = 10, n = 1000000 and d = 1: The average of 10 run times is 31.3 ms
- For m = 10, n = 2500000 and d = 1: The average of 10 run times is 60.2 ms

---

Now we test the same for bigger values of m and check the efficiency of the code.

We will test edge and middle cases from here on out and draw comparison between the naiv and the improved insertion sort

- For m = 25, n = 10000 and d = -1: The average of 10 run times is 101.4 ms
- For m = 25, n = 100000 and d = -1: The average of 10 run times is 598.14ms
- For m = 25, n = 500000 and d = 1: The average of 10 run times is 148,705 ms
- For m = 25, n = 1000000 and d = 1: This takes longer than 10 mins to run so we're stopping the simulation here.

If we compare this to the earlier algorithm we can find out that the runtimes are significantly less for higher value inputs of n and m.
In the previous algorithm it took more time to process an input of n = 250000 as compared to this algorithm which runs faster even for inputs like 500000. This is what evinces the decreased time complexity of the improved insertion sort algorithm.

- For m = 25, n = 10000 and d = 0: The average of 10 run times is 39.66 ms
- For m = 25, n = 100000 and d = 0: The average of 10 run times is 2,208.5ms
- For m = 25, n = 500000 and d =0 : The average of 10 run times is 65,924.3 ms
- For m = 25, n = 1000000 and d = 0: This takes 266,570 ms to run (average of 5 times) as this takes about 10 mins to execute in real time. Hence we are stopping the simulation here
- For m = 25, n = 2500000 and d = 0: This has very high run time and hence needs merge sort to efficiently go through higher m and n values.

Now we test for an already sorted array which is the best case scenario.

- For m = 25, n = 10000 and d = 1: The average of 10 run times is 0.3 ms
- For m = 25, n = 50000 and d = 1: The average of 10 run times is 1.3 ms
- For m = 25, n = 100000 and d = 1: The average of 10 run times is 5.7 ms
- For m = 25, n = 500000 and d = 1: The average of 10 run times is 14.8 ms
- For m = 25, n = 1000000 and d = 1: The average of 10 run times is 31.3 ms
- For m = 25, n = 2500000 and d = 1: The average of 10 run times is 60.2 ms

Now we test for higher m values

Now we test for the value on m = 50 and different values of n and d

- For m = 50, n = 10000 and d = -1: (48+49+47+48+48+48+47+47+48+47)/10=47.7ms
  The average of 10 run times is ms
- For m = 50, n = 100000 and d = -1:
  (4866+4847+4846+4853+4845+4841+4855+4856+4856+4858)/10=4,852.3ms
  The average of 10 run times is ms

- For m = 50, n = 500000 and d = -1: 18,772.6ms
  (146852+146087+146766+146817+146075)/5=146,519.4ms
  The average of 10 run times is ms
- For m = 50, n = 1000000 and d = 1: This takes longer than 10 mins to run in real time hence we're stopping the simulation here.

Here we test for the worst case condition for insertion sort i.e when the input array is reverse sorted. We test it for the above values of n and we can see the run times increase in the order of O(n^2) as the input size and the vector size increases. We soon reach that 10 minute threshold and such datasets are better suited for merge sort as we can see from later parts of this report.

Here we test a randomly sorted array and feed it into the insertion sort function. It's time taken is completely dependant on the order of inputs of the randomly generated array. The run time also increases for higher values of m and n. And we stop the simulation after it reaches that 10 minute threshold.

- For m = 50, n = 10000 and d = 0: (25+24+27+25+27+25+25+27+27+30)/10= 26.2ms
  The average of 10 run times is ms

- For m = 50, n = 100000 and d = 0:
  (2392+2394+2404+2497+2410+2397+2399+2397+2398+2401)/10 = 2,408.9ms
  The average of 10 run times is ms

- For m = 50, n = 500000 and d =0 :
  (67428+66638+66993+66378+66406+67142+66813+66399+66810+66421)/10=66,742.8ms

- For m = 50, n = 1000000 and d = 0: (314619+312950)/2=313,784.5ms

- For m = 50, n = 2500000 and d = 0: This takes about 10 mins to execute in real time. Hence we are stopping the simulation here.

Here we test for the best case scenario where a sorted array is fed into the insertion sort algorithm. This in either insertion sort algorithm obtains similar results as this is the best case scenario.

- For m = 50, n = 10000 and d = 1: (1+0+0+0+0+0+1+1+0+1)/10=0.4ms
  The average of 10 run times is ms

- For m = 50, n = 50000 and d = 1: (6+2+3+2+6+2+2+3+3+1)/10=3ms
  The average of 10 run times is ms

- For m = 50, n = 100000 and d = 1: (3+3+4+3+3+3+3+3+3+3)/10=3.1ms
  The average of 10 run times is ms

- For m = 50, n = 500000 and d = 1: (14+15+14+16+14+15+14+16+14+21)/10=15.3ms
  The average of 10 run times is ms

- For m = 50, n = 1000000 and d = 1: (29+29+29+29+29+30+35+29+34+34)/10=30.7ms
  The average of 10 run times is ms

- For m = 50, n = 2500000 and d = 1: (88+80+79+78+77+78+78+79+82+98)/10 =81.7ms
  The average of 10 run times is ms

MERGE SORT

- For m = 10, n = 10000 and d = -1: (22+22+20+23+22+20+24+23+23+22)/10=22.1ms
- For m = 10, n = 25000 and d = -1: (56+57+52+56+53+53+52+62+52+53)/10=54.6ms
- For m = 10, n = 50000 and d = -1: (108+108+111+109+113+109+111+110+109+111)/10=109.9ms
- For m = 10, n = 100000 and d = -1: (229+224+227+224+228+227+228+228+224+228)/10=226.7
- For m = 10, n = 250000 and d = -1: (593+590+593+596+609+608+604+602+606+598)/10=599.9
- For m = 10, n = 500000 and d = -1: (1269+1264+1250+1247+1255+1260+1246+1361+1469+1474)/10=1,309.5ms
- For m = 10, n = 1000000 and d = -1: (2567+2567+2602+2671+3085+3059+3071+3022+2998+3084)/10=2,872.6ms
- For m = 10, n = 2500000 and d = -1: (6773+7443+8252+8226+8374+8406+8526+8469+8704+8543)/10=8,171.6ms

If we compare this to the insertion sort algorithms , this merge sort algorithm is more efficient for larger data values as its time complexity is $\Theta(nlogn)$ as the time complexity of insertion sort is O(n^2) the run times of merge sort is much faster than insertion sort for larger m and n values.

- For m = 10, n = 10000 and d = 0: (21+24+24+22+23+29+27+24+20+23)/10=23.7ms
- For m = 10, n = 25000 and d = 0: (55+60+54+56+57+55+55+59+59+55)/10=56.5ms
- For m = 10, n = 50000 and d = 0: (113+115+116+114+114+110+111+113+114+113)/10=113.3ms
- For m = 10, n = 100000 and d = 0: (231+229+232+231+229+230+233+230+229+230)/10=230.4ms
- For m = 10, n = 250000 and d = 0: (601+599+604+607+602+603+608+603+608+603)/10=603.8ms
- For m = 10, n = 500000 and d = 0: (1248+1244+1247+1254+1254+1259+1470+1471+1271+1329)/10=1,304.7ms
- For m = 10, n = 1000000 and d = 0: (2613+2622+2611+2643+2832+3291+3350+3297+3185+3428)/10=2,987.2ms
- For m = 10, n = 2500000 and d = 0: (6910+7192+8795+9032+11309+8604+8898+9037+9171+9899)/10=8,884.7ms

These inputs are randomly generated array inputs of size n and m as specified above, and these can be created in any random order. The logs above show the time taken for each of the specified inputs. And that this algorithm works efficiently for a randomly generated input array of size 2500000.

Input here is a sorted array and hence the following are the runtimes

- For m = 10, n = 10000 and d = 1:(24+22+21+23+29+23+22+19+22+23)/10 = 22.8ms
- For m = 10, n = 25000 and d = 1: (52+54+55+53+57+53+55+54+53+59)/10 = 54.5ms
- For m = 10, n = 50000 and d = 1: (111+109+108+110+113+110+109+108+112+112)/10 =110.2ms
- For m = 10, n = 100000 and d = 1: (237+223+222+229+222+223+222+224+222+224)/10=224.8ms
- For m = 10, n = 250000 and d = 1: (591+586+588+587+594+587+593+591+585+591)/10=589.3ms
- For m = 10, n = 500000 and d = 1: (1217+1227+1213+1218+1219+1217+1218+1233+1240+1250)/10=1,225.2ms
- For m = 10, n = 1000000 and d = 1: (2574+2560+2553+2580+2586+3104+3029+3022+3099+3020)/10=2,812.7ms
- For m = 10, n = 2500000 and d = 1: (6795+6897+8071+8097+8448+8258+8600+8448+8530+8674)=8,081.8ms

The above test statements create a presorted array which is then passed to the merge sort algorithm . This then runs at a speeds not so different for the insertion sort algorithm. As this is the best case scenario for insertion sort algorithm it runs as fast. Since merge sort has the same time complexity for best case, worst case and average case i.e $\Theta(nlogn)$. It doesn't see too much change in either one of those cases.

For Different values of m i.e m= 25 and m = 50

Input here is a reverse sorted array and hence the following are the runtimes

- For m = 25, n = 10000 and d = -1:(30+31+30+27+28+26+27+30+31+35)/10=29.5ms
- For m = 25, n = 50000 and d = -1:
  (142+139+142+141+142+140+139+144+147+144)/10=142ms
- For m = 25, n = 250000 and d = -1:
  (770+766+770+766+768+770+770+788+787+789)/10=774.4ms
- For m = 25, n = 500000 and d = -1:
  (1598+1595+1598+1650+1651+2186+2106+2138+2104+2088)/10=1,871.4ms
- For m = 25, n = 2500000 and d = -1:
  (9345+9504+9325+9261+9246+9223+9312+9785+9432+9344)/10=9,377.7ms

Above logs the results of a higher vector size of 25 and tests it with several values of input arrays n , Now in this a reverse sorted array is generated and put into the merge sort algorithm, since this is an average case for merge sort and also since merge sort handles bigger data, better we see low runtimes as seen in the results above as compared to either of the insertion sort algorithms.

- For m = 25, n = 10000 and d = 0: (27+29+29+28+32+33+30+28+28+32)/10=29.6ms
- For m = 25, n = 50000 and d = 0:
  (145+148+153+151+146+155+148+159+146+170)/10=152.1ms
- For m = 25, n = 250000 and d = 0:
  (778+784+800+790+780+792+795+794+800+802)/10=791.5ms
- For m = 25, n = 500000 and d = 0:
  (1614+1623+1624+1674+1767+2525+2386+2337+2433+2384)/10=2,036.7ms
- For m = 25, n = 2500000 and d = 0:
  (9505+9537+9532+9526+9551+9491+9588+9512+9559+9591)/10=9,539.2ms

Above logs the results for a higher value of m =25 and multiple values of input array n. Now in this a randomly generated array is inserted into the merge sort algorithm and as we can see , since merge sort has just an average case i.e $\Theta(nlogn)$. Due to this if we compare the run

times for the reverse generated array and the randomly generated array for merge sort, we find that they are very similar as they are treated the same.

- For m = 25, n = 10000 and d = 1: (27+31+29+26+32+33+30+29+28+31)/10=29.6ms
- For m = 25, n = 50000 and d = 1: (145+144+153+151+146+151+148+159+143+171)/10=151.1ms
- For m = 25, n = 250000 and d = 1: (778+784+800+790+780+792+795+794+800+802)/10=788.3ms
- For m = 25, n = 500000 and d = 1: (1614+1623+1624+1674+1767+2525+2386+2337+2433+2384)/10=2,026.4ms
- For m = 25, n = 2500000 and d = 1: (9845+15044+14742+15898+17418+16152+18421+16818+16353+15126)/10=16,581.7 ms

Above logs the results of a higher value of m =25 and multiple values of input array n. Now we feed a fully sorted array as input into the merge sort algorithm and from the logged values above we can confirm that merge sort treats every input array the same , it doesn't have a best case or a worse case. We can say that insertion sort is better for handling smaller data as compared to merge sort but, merge sort is better for processing large datasets.

For value of m =50

- For m = 50, n = 10000 and d = -1: (33+37+35+37+38+38+32+35+38+33)/10=35.6ms
- For m = 50, n = 50000 and d = -1: (192+189+190+192+187+194+187+188+190+189)/10=189.8ms
- For m = 50, n = 250000 and d = -1: (1030+1023+1016+1048+1051+1111+1455+1464+1431+1463)/10=1,209.2ms
- For m = 50, n = 500000 and d = -1: (2275+2160+2151+2163+2153+2156+2179+2157+2162+2155)/10=2,171.1ms
- For m = 50, n = 2500000 and d = -1: (15766+15738+15658+15440+15851+15324+15481+15419+15560+15664)/10=15,590. 1ms

We now feed a higher value of m=50 into the merge sort algorithm. Now this is tested for multiple input arrays of size 10000, 50000, 250000, 500000 and 2500000. And we find out that it has a longer running time than if we insert a lower value of vector m i.e m=25 , but we can still see that this is more efficient for processing larger datasets .

Here we use a randomly generated array and pass it to the merge sort function. And we can see the runtimes for these inputs are almost similar to when we passed the inputs in the previous section i.e(For m = 50, n = 10000 and d = -1 ……. For m = 50, n = 2500000 and d = -1) and this is again due to the fact that merge sort treats all input arrays the same . Merge sort splits the given array in half and then recursively calls itself . It then calls the merge function and hence you have the sorted output. This is why it treats every input the same.

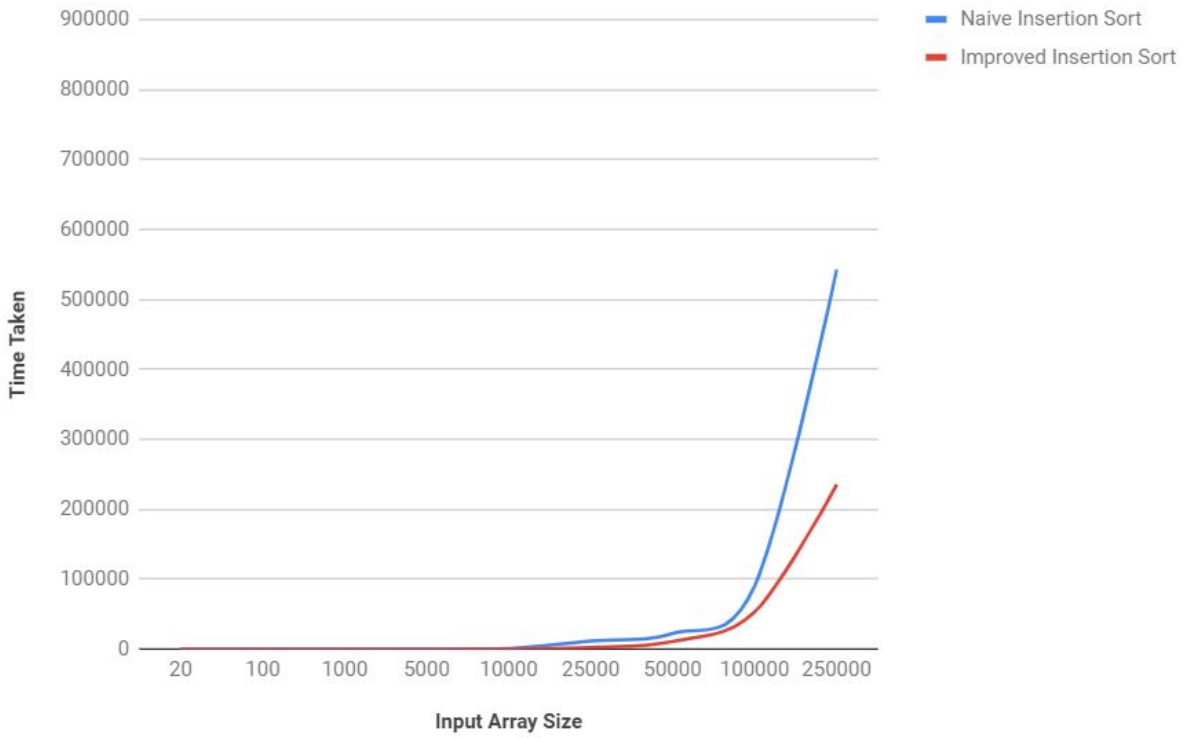
- For m = 50, n = 10000 and d = 0: (36+40+39+36+37+38+39+37+43+40)/10=38.5ms
- For m = 50, n = 50000 and d = 0:
  (192+190+191+193+198+198+192+194+197+190)/10=193.5ms
- For m = 50, n = 250000 and d = 0:
  (1049+1045+1051+1044+1044+1046+1048+1044+1045+1047)/10=1046.3
- For m = 50, n = 500000 and d = 0:
  (2206+2197+2197+2198+2182+2207+2197+2192+2189)/10=1,976.5ms
- For m = 50, n = 2500000 and d = 0:
  (21050+20930+21085+20295+20472+20920+20427+21071+20518)/10=18,676.8ms

---

This is when we input a fully sorted array input of the following parameters :

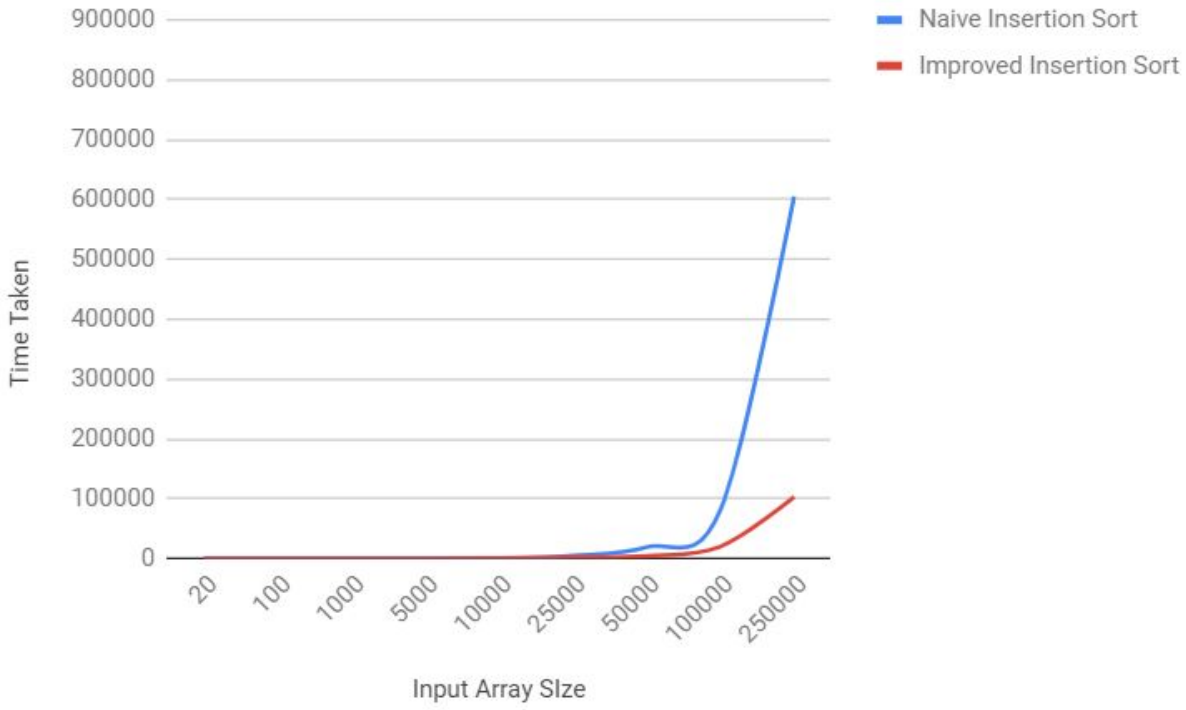
- For m = 50, n = 10000 and d = 1: (40+42+43+36+39+38+41+38+37+39)/10=39.3ms
- For m = 50, n = 50000 and d = 1:
  (193+190+191+193+191+187+193+194+188+193)/10=191.3ms
- For m = 50, n = 250000 and d = 1:
  (1029+1029+1036+1034+1032+1033+1027+1034+1030+1033)/10=1,030.7ms
- For m = 50, n = 500000 and d = 1:
  (2267+2176+2181+2162+2160+2161+2164+2173+2217+2267)/10=2,192.8ms
- For m = 50, n = 2500000 and d = 1:
  (15694+15693+15496+15351+16279+16101+16184+15446+15350+15399)/10=15,699.3ms

Comparisons

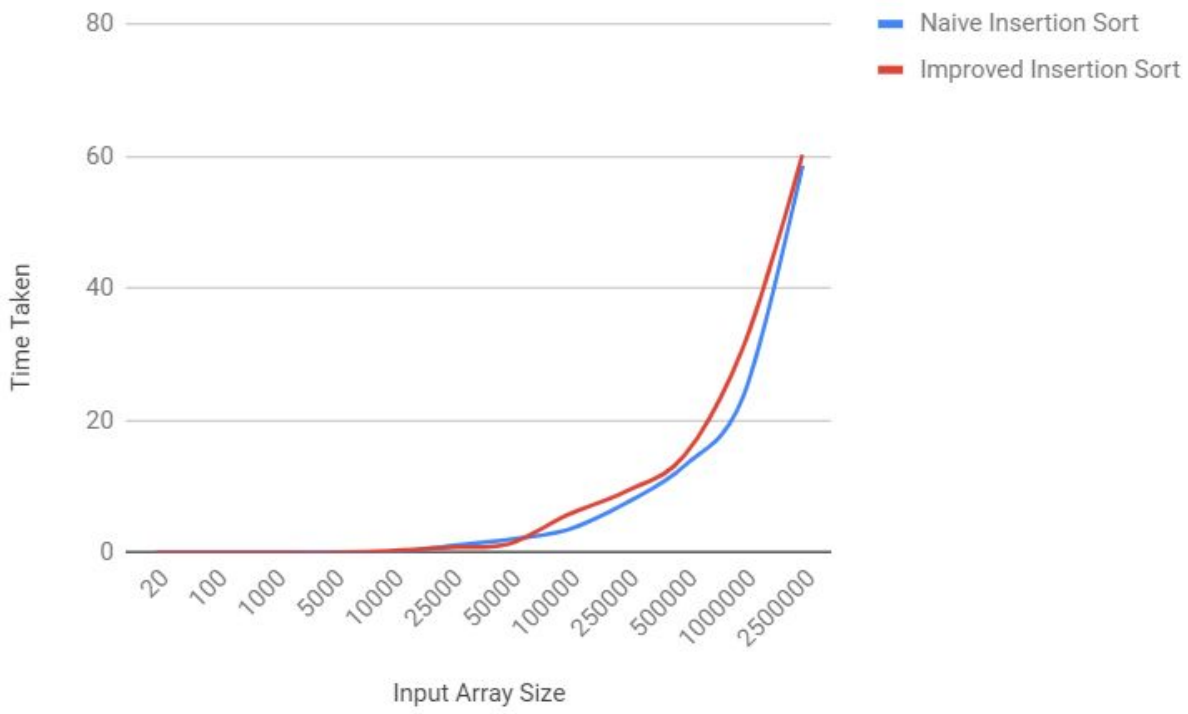


In the above graphs we compare the average time taken for naive insertion sort algorithm vs the improved insertion sort algorithm. We notice that the runtimes are about the same when it has smaller input sizes of n , in the order of 10 , 20 , … But as we start increasing the input size the Improved insertion sort algorithm proves to be more efficient. The above was tested for reverse generated array inputs i.e d = -1 and the value of m =10.

Now we take a look at the comparison graphs for randomly generated arrays as input i.e d = 0 and the value of m = 10.

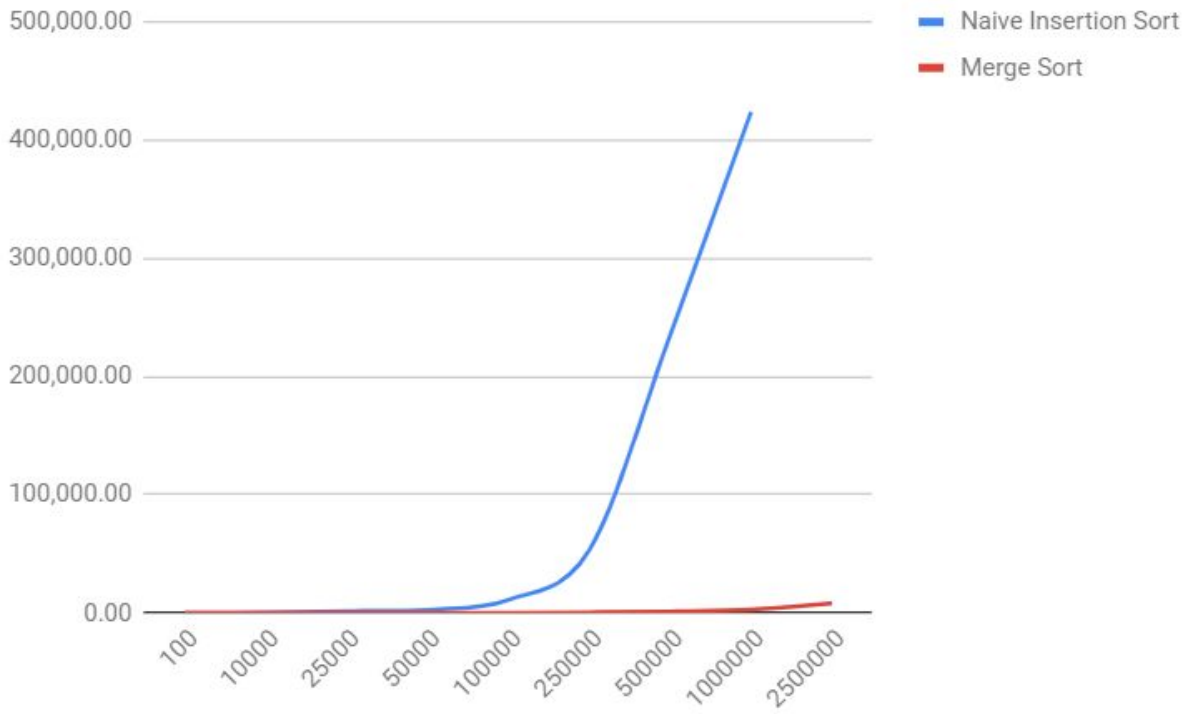


Now we take a look at the comparison graphs for sorted arrays as input i.e d = 1 and the value of m = 10



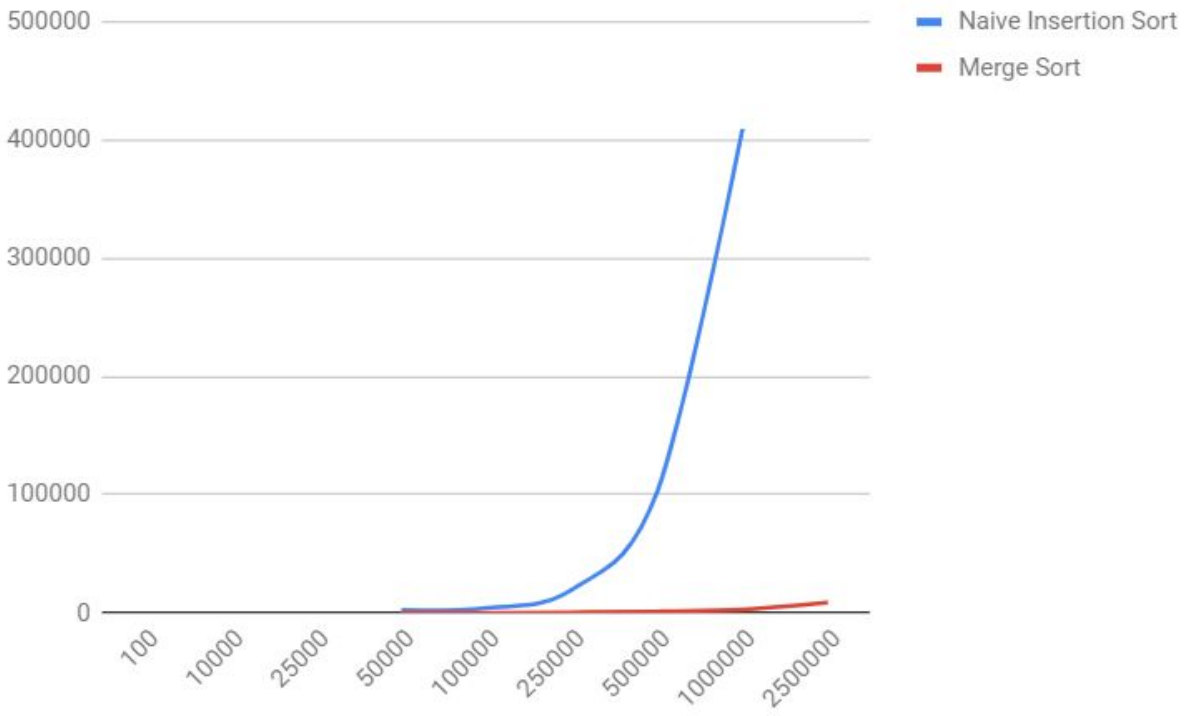
Here we find out that when the input array is fully sorted, then the runtimes for both the naive and the improved insertion sort are fairly similar.

Now we compare the improved insertion sort vs merge sort.
We consider m value = 10 and d value = -1 (reverse sorted array input)



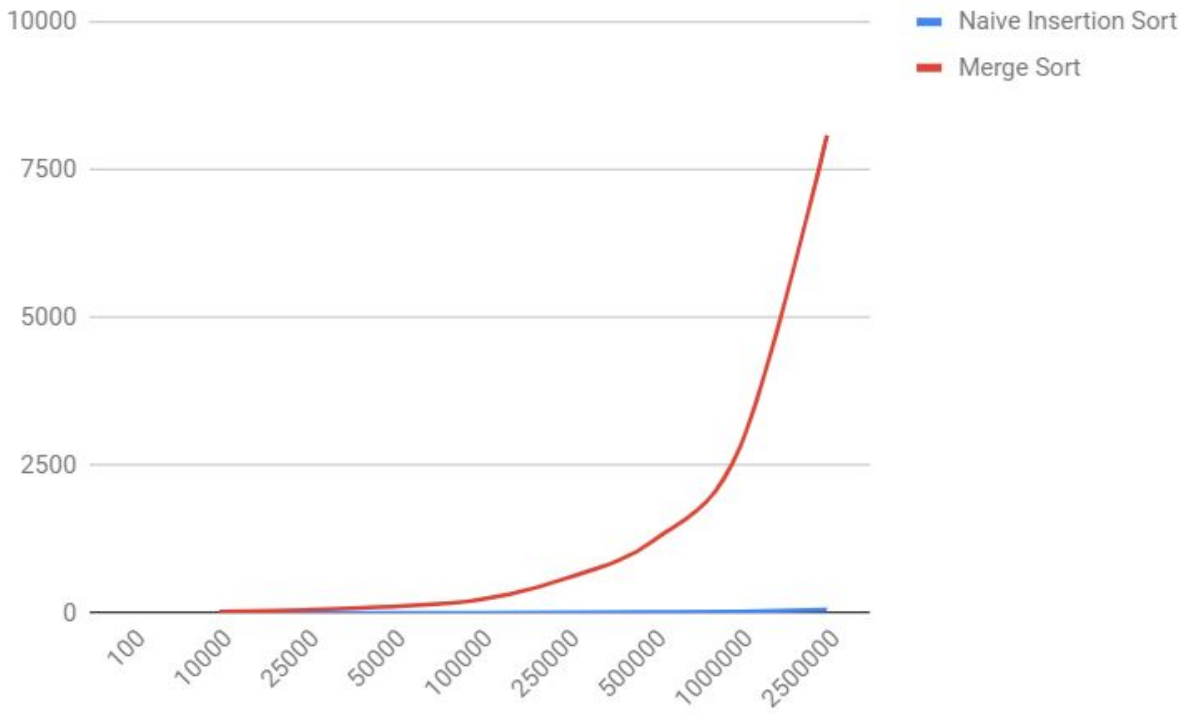
The above graph proves how much more efficient the merge sort algorithm is in dealing with larger datasets. Now we check the graphs for the input array being (randomly generated and fully sorted) .

Now we consider m value = 10 and d value = 0 (randomly sorted array)



Even when the input array is randomly sorted the merge sort algorithm enjoys faster run times.

Now we consider m value = 10 and d value = 1 (fully sorted array)



Here we see that for a sorted array which is the best case scenario in insertion sort, it is more efficient to use insertion sort than merge sort in this case of sorted data. This is due to the fact that merge sort has no best case or worst case. It only has average case.

We conclude by listing our learnings:

1. Insertion sort has a time complexity of O(n^2) and that the worst case scenario for insertion sort is when a reverse sorted array is input and the best case is when a fully sorted array is passed to the algorithm as input.
2. Insertion sort can also have a time complexity of O(n^3) when calling ivector_length 2 times.
3. Merge sort has a time complexity of $\Theta(nlogn)$. and that its best case , average case and worst case is all the same.
4. Due to Merge sort having the same best, average, and worst case. We noticed that even for larger data sets, it doesn't matter what sorting order the input is provided i.e whether its reverse sorted, random or fully sorted . It has similar runtimes for most of these cases.
5. Insertion sort is better for smaller input size.
6. Merge sort is better suited for larger datasets .

==========END==========