Joseph Vetri

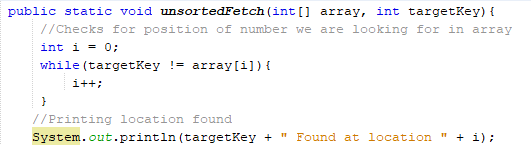
Assignment 2

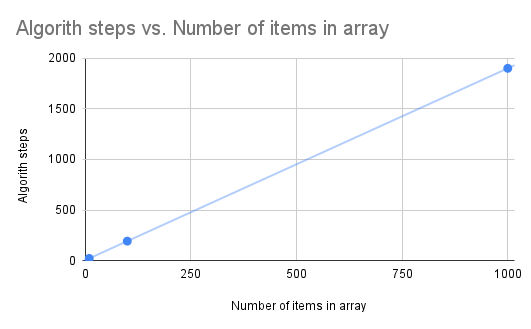
COM210

Lab Report

**UNSORTED FETCH**

The first algorithm I wrote was the unsorted fetch. This algorithm has a big O of O(n) since it is a linear algorithm.





With a small array of 10 elements the algorithm took 21 steps to complete including all statements inside the method. The medium sized array took 192 steps to complete. And the large array took 1902 steps to complete. This proves that the algorithm is O(n) because the number of steps is getting proportionally larger in regards to the number of items in the array.

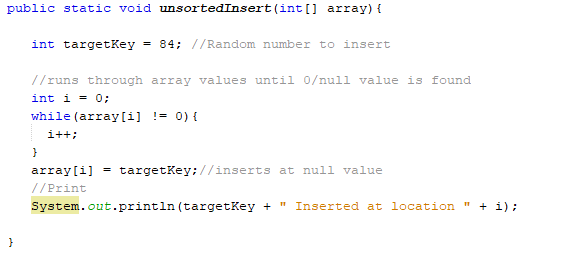
I included the print statements and declarations in the step count. Without them it would be 19 steps, 190 steps, and 1900 steps.

The value being searched for is the worst possible case, which is 10-9 , 100-99, 1000-999.

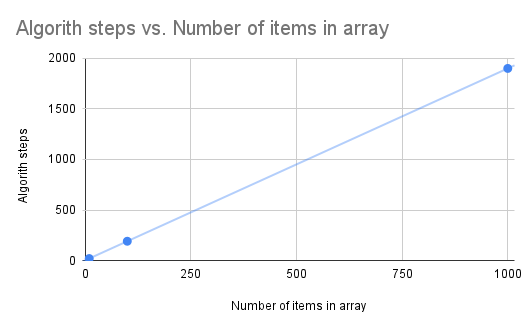
The worst case is the highest in unsorted algorithms because it is searching linearly.

**UNSORTED INSERT**

The second algorithm I wrote was an unsorted insert. I am fairly certain the time complexity of my algorithm is O(N). When it should really be O(C)?



I believe this is due to the fact that my algorithm is searching for a null/0 location and not assuming one at the end.



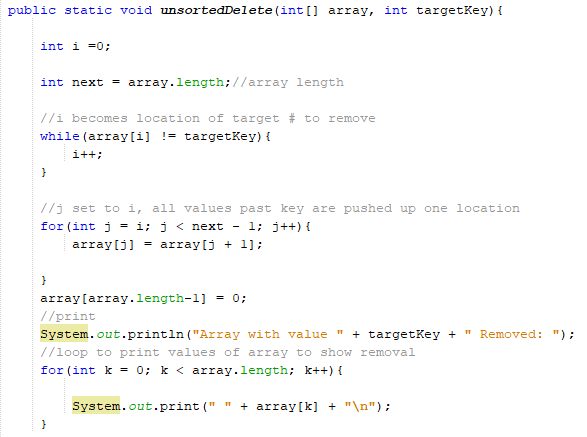
For 10 elements this algorithm took 20 steps. It is 20 because I included 3 null values into the 10 element array. It finds the first 0 at location 7 and not 9. I did this originally because I had all the algorithms using the same arrays, but, I changed that because it was confusing to understand if the algorithms worked due to manipulating the same arrays over and over.

For 100 elements it took 191 steps, and for 1000 it took 1901 steps. That makes this an O(N) algorithm.

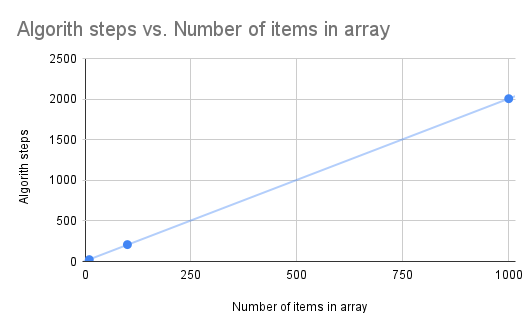
Each iteration is looking for the worst case scenario, in the 10 element array it is 9, 100- 99… etc.

**UNSORTED DELETE**

This is an O(N) algorithm along with all the other unsorted algorithms. This is because as the number of items in the array increases the algorithm steps increase proportionally. In the code below the while loop searches for the targetkey and the for loop is pushing all values past the key up one location.

****

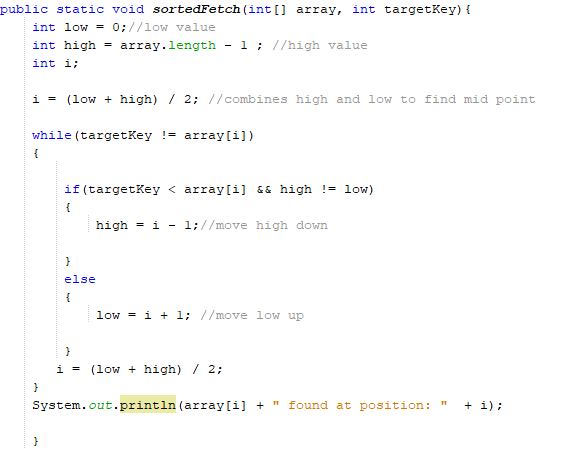
I did not include the for loop at the bottom in the step counting because it is completely irrelevant, and I did not include the print statement above the for loop aswell.

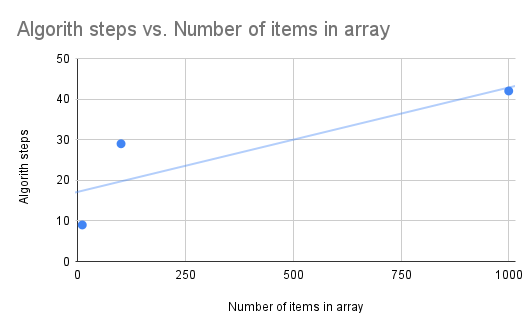
****

For 10 elements this algorithm took 23 steps, for 100 this algorithm took 208 steps and for 1000 elements this array took 2005 steps. This algorithm is O(N) because it is increasing linearly. The value I used to delete in each array was 1, which is the smallest value, and means that every value after it will have to shift in order for 1 to be removed.

**SORTED FETCH**

This algorithm uses a binary search to find the location of the desired value.



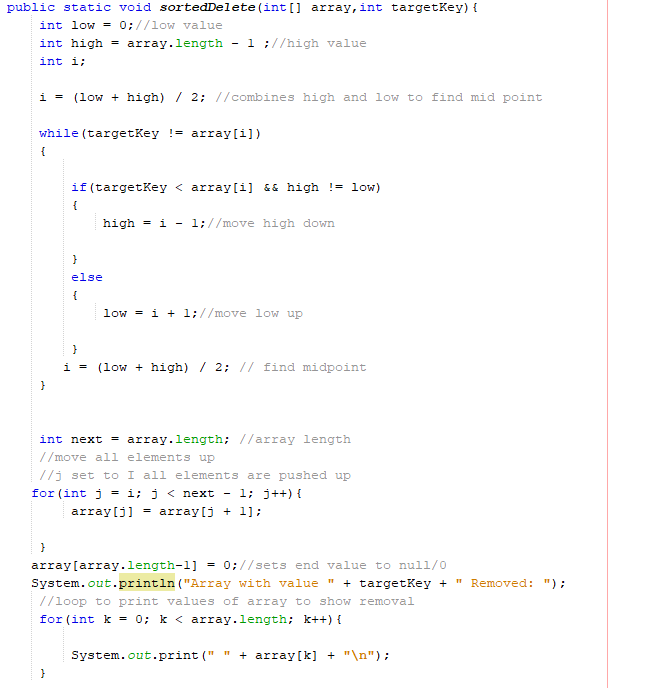


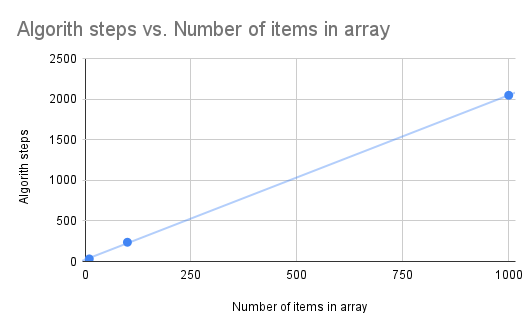
For 10 values this algorithm took 9 steps, for 100 it took 29, and for 1000 it took 42 steps. This algorithm is O(Log N) , Compared to the other fetch algorithm this is much more efficient. The binary search is efficient due to its ability to split the array over and over, lowering the amount of steps it takes to find the value.

I fetched the worst case value (1) to count the number of steps.

**SORTED DELETE**

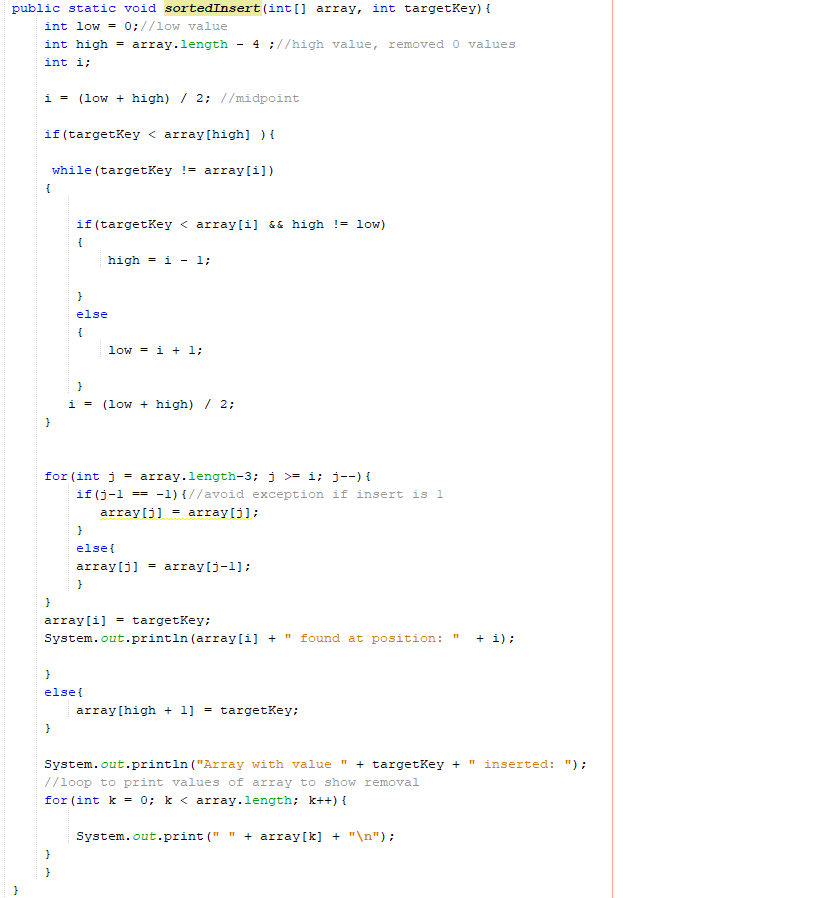
The sorted delete uses a binary search to fetch the value we want to delete, the binary search is O(log n) but due to the fact that we have to delete by pushing all elements up the final big O becomes O(N) .



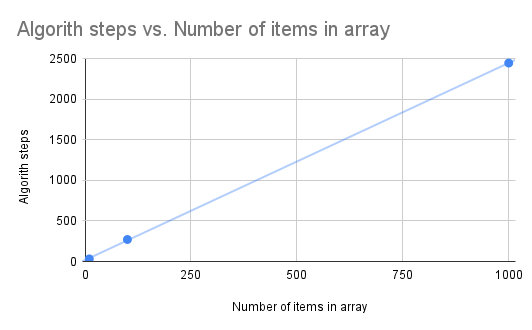


This chart is reminiscent of the unsorted algorithms. This is because I chose the worse case value for each array, which is 1. Due to the deleting part of the algorithm when the value is one, each element in the array has to be shifted. 10 elements took 32 steps, 100 took 237 and 1000 took 2047 steps. This algorithm has a big O of O(N) due to the shifting of the elements.

**SORTED INSERT**

****

This algorithm uses a binary search to find the location of insert which is O(log n) but because just like the delete algorithm a linear search is used to shift all the elements down. This will make the big O of the algorithm O(N).



This algorithm follows a linear path. The binary search algorithm prevents excessive searching but the shift is not efficient. I used 1 again as the worst case scenario for all inserts. 10 elements took 32 steps, 100 took 271 and 1000 took 2444.

**CONCLUSIONS**

Theoretically these algorithms are supposed to be efficient but when presented against the worst case scenario they do not perform too well. Due to some differences between my code and the most efficient implementation of these algorithms the number of steps must be slightly worse than it could be if I didn't do it myself.

The sorted fetch is the best example of an efficient algorithm out of all of them, it is O(log n) and it takes less steps as the number of elements in the array gets larger.

Big O notation is how programmers figure out the complexity of an algorithm. My experimentation with these algorithms has presented me with some clarity on it.