1.1 - Checks if the array is of length one, if it is, checks if the first element is zero, if the element is 0 returns true, if the list is bigger than one element it breaks it into 3 sub lists and calls itself recursively.

1.2 - n = length(x)
T(n) = T(1) + T(n/3) + T(n/3) + T(n/3)

1.3 -

T(n) = T(1) + T(n/3) + T(n/3) + T(n/3)
T(n) = T(1) + 3(T(n/3))
an + b = c + 3(a(n/3)+b)
an + b = c + 3an/3 + 3b
~~an~~ + b = c + ~~an +~~ 3b
b = -c/2

T(n) = T(1) + 3(T(n/3))
= c + 3(a(n/3) + b)
= c + an + 3b
= c + an + 3(-c/2)
= an + c_2
-> O(n)

1.4 -

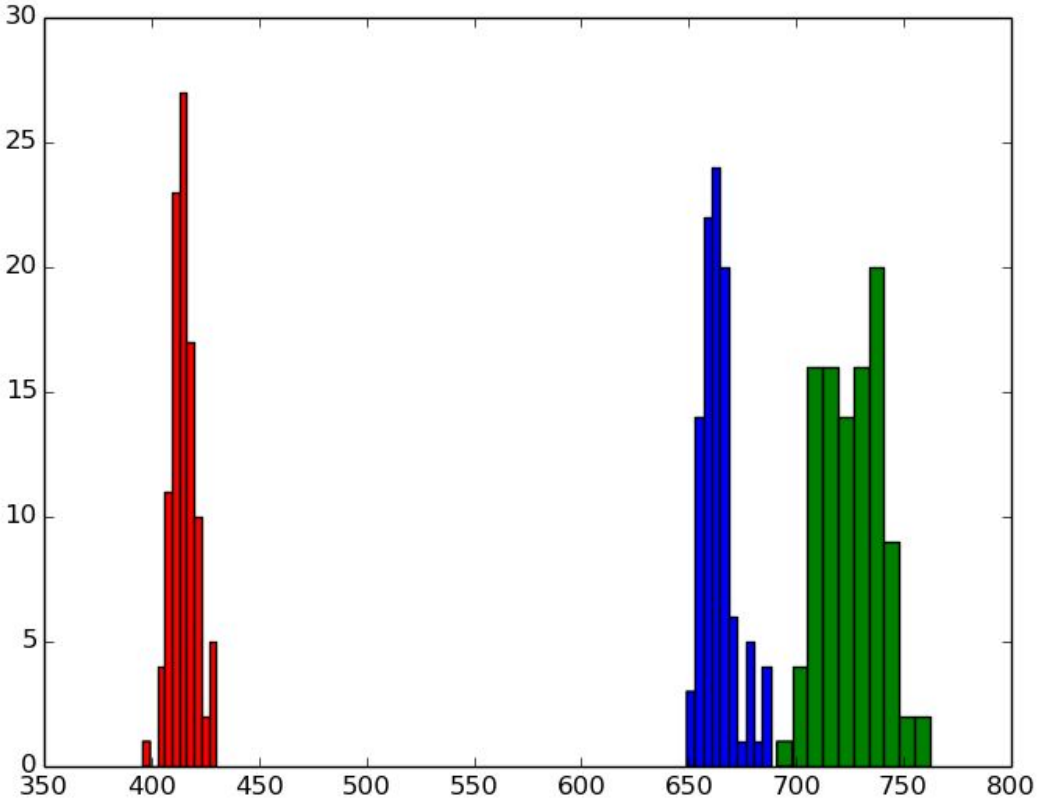
```
for(int i = 0; i < length; i++)
{
        if(list[i] == 0){return true;}

        return false;
}
```

1.5 - Both recursive and iterative have an asymptotic runtime complexity of O(n).

2.1 - In Java each character requires 2 bytes of memory, you have 100 characters and 1 million strings. Therefore the memory required is 100(chars)*2(bytes/char)*1000000(strings) = 200000000 bytes/2^{30}(bytes/GB) = 0.186264514923096 GB

2.3 - The histogram shows that quicksort is faster than heapsort which is faster than merge sort.



3.1 - Assume all characters are in the ASCII set, therefore only one byte required to store them. There are 102 characters per line(100 real characters plus the LF and CR characters), and there are 100 million of them. 100M*102 = 10,200,000,000 bytes which equates to 9.5GB.

3.3 - The algorithm I have designed works by first generating 100 million strings and storing them in a file. Then it reads 1 million in at a time to an array and merge sorts them and saves them in their own files. It then generates buffered readers for each of the 100 files that were created in the first step. It reads in one line from each of the files into an array of size 100. It the iterates through that array and finds the smallest element. It appends that element to a new file and reads in a new element from the same file that the smallest element came from. I repeats this process until all elements have been appended to the new file.

3.4 - Merge sorting the individual chunks of data is $n \log_2 n$, then the merge operation has to operate over the entire array of buffered readers which is n/(number of splits) and it has to be done n times. This results in a complexity of n^2/(number of splits) + $n \log_2 n$. Since the number of splits is a constant and we only carry about the most dominant function the function is big O of

n^2. Using the definition of data pass from the Wikipedia article on external sorting [1], this algorithm must make 2 passes on the data. The first to read it into chunks and sort it. The second to compare the values in the array and merge them into one sorted file.

3.5 - The runtime of the algorithm was 1093116ms or 1093.116s which equates to 18.217 minutes.

1. https://en.wikipedia.org/wiki/External_sorting#Additional_passes