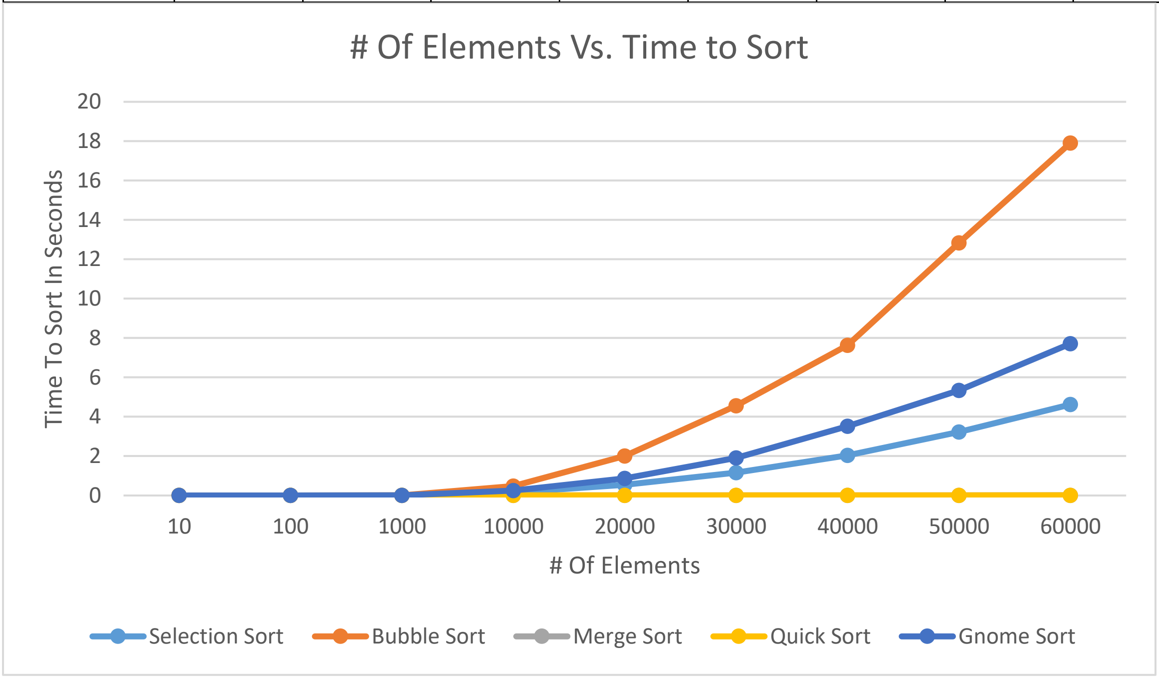
Andrew Tran

Sorting Algorithm	*Dataset 10*	*Dataset 100*	*Dataset 1000*	*Dataset 10000*	*Dataset 20000*	*Dataset 30000*	*Dataset 40000*	*Dataset 50000*	*Dataset 60000*
Selection Sort	0 s	0 s	0.002 s	0.145 s	0.539 s	1.156 s	2.043 s	3.230 s	4.621 s
Bubble Sort	0 s	0 s	0.004 s	0.471 s	2.000 s	4.555 s	7.635 s	12.831 s	17.909 s
Merge Sort	0 s	0 s	0 s	0.002 s	0.004s	0.006 s	0.008 s	0.010 s	0.0120s
Quick Sort	0 s	0 s	0 s	0.001 s	0.003s	0.005 s	0.007 s	0.009 s	0.0110s
Gnome Sort	0 s	0 s	0.002 s	0.245 s	0.864 s	1.910 s	3.514 s	5.355 s	7.712 s



Sorting Algorithms	*Worst*	*Average*	*Best*
Selection Sort	O(n^2)	O(n^2)	O(n^2)
Bubble Sort	O(n^2)	O(n^2)	O(n)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)/O(n)
Quick Sort	O(n^2)	O(nlogn)	O(nlogn)/O(n)
Gnome Sort	O(n^2)	O(n^2)	O(n)

Selection Sort:

Selection sort is a sorting algorithm that uses the in-place comparison sort method. This means that during the sorting process, no auxiliary data structures are used to sort an inputted dataset. This is done by having the sorted output overwrite the original index positions of the inputted array. During the process of this sorting algorithm, the array will have two sections in it, the sorted and unsorted sections of the array. According to the table above, the running time for best, average, and worst is $O(n^2)$. This is true since for either of the 3 running times, due to selection sort always needing to run the same number of times as the number of elements in the array. Therefore, selection sort will not be the best choice for sorting large datasets. This can be seen with the for loop below which is used in selection sort:

```
  for(i =0;  i<n-1;  i++)
    {
//rest of code in c++ file
    }
```

The loop will execute (n-1) when i =0 then (n-1)+(n-2) times for i=1 and then this will be an arithmetic progression leaving us with the equation of [n(n-1)]/2 which simplifies to $[n^2-n]2$. In big O notation, all that is cared about is the highest order term of a polynomial which is where the run time of big $O(n^2)$ comes from for the sorting algorithm.

How selection sort works:

Selection sort will have two lists; an unsorted list and a sorted list. The sorted list will start off empty, but as the algorithm progresses through to the end the unsorted list will become empty and the sorted list will be full. The algorithm automatically sets the value at index 0 as the smallest element in the array and will have a for loop that will loop until n-1 times. Then there is another inner for loop that will loop from whatever index the outer for loop is one is at plus one. In the inner for loop, there will be an if statement that will compare whatever the value at the index the outer for loop is on with whatever the value at the index of the inner loop is on and if the value of the inner loop is smaller than the value of the outer loop, the value of the inner loop becomes the new minimum. Then the loop exits and checks to see if the new minimum is in the correct index, if it is not swap the index of the outer loop with the inner loop. The new minimum goes to the left side and the old minimum goes to the right side of the array. Then the outer loop will advance. If the value at the index of the inner loop is greater than the value at the index of the outer loop, do not swap anything and continue to advance the inner loop. The final value will not loop since they algorithm knows the last value will always be in order by the end of the algorithm.

The cases:

Overall, selection sort does not have a best, average or worst running time since all the running times have identical big O notations of O(n^2). Since the outer loop is always the arithmetic equation of [n^2-n]/2, whatever dataset that is inputted in the algorithm will always have to loop n-1 times no matter how big or small the data set is.

Other facts about Selection Sort:

Although, Selection Sort may seem to be slow with a big O notation of O(n^2), it still outperforms Bubble Sort all the time and outperforms Merge Sort and Quicksort in smaller datasets. Selection Sort is also very bad choice for large datasets because it requires looping through each dataset n-1 times the number of elements in the array. Another fact is that Selection Sort also underperforms Insertion Sort which is a sorting algorithm that is similar to Selection Sort.

Example of a Selection Sort run:

70 12 5 6 90 //initial input

5 12 70 6 90 //after 1 loop sorted sublist {5}

5 6 70 12 90 //after 2 loops sorted sublist {5 6}

5 6 12 70 90 //after 3 loops sorted sublist{5 6 12}

5 6 12 70 90 //after 4 loops sorted sublist{5 6 12 90}

5 6 12 70 90 //after 5 loops sorted sublist{5 6 12 90}

Bubble Sort:

Bubble Sort is a fairly simple sorting algorithm and is also known as a **sinking sort** algorithm because the largest element sink to the end of the list until the list is fully sorted.

How Bubble Sort works:

Bubble Sort continuously moves through an array with an outer for loop to loop as many times as there are elements in the array, but there is a special feature that checks if there are no more swaps to be done the loop exits increasing efficiency of the algorithm. Then there is an inner for loop that does the swapping by checking for example if the value at index 0 which is the current index value is bigger than the value at index 1 which is the right adjacent index value, if the value at index 1 is bigger, swap the two values, if advance the current index and perform the next comparison. The next comparison will be between the value at index 1 and the value at index 2. The same thing happens; if the value at index 1 is bigger than the value at 2 swap the two values. This continues until the end of the array is reached and after one loop of the outer loop, the largest value will be at the end of the array. If the array is sorted earlier than n times elements, the outer for loop will exit and the list will be fully sorted.

Other facts about Bubble Sort:

Bubble sort appears to be the slowest sorting algorthim out of the 5 in this assignment which can be seen with the experimental run times along with the graph. Adding on to this, Bubble Sort also has a big O notation of O(n^2) where n represents the number of elements to sort. Knowing this, there are also sorting algorithms that exist with worst, average, and best Big O notations of O(n log n) such as Merge Sort that outperform Bubble Sort and other O(n^2) such as Insertion sort and Selection Sort (which can be seen in graph and table) that outperform Bubble Sort. Therefore, it proves that Bubble Sort is not the best choice for sorting large datasets since it is very slow. Although Bubble Sort is very slow, it has one special feature that some other algorithms do not have not even Quicksort has it where the algorithm stops looping when it goes through and recognizes that the whole dataset is sorted. Quicksort does not even have this feature and it is one of the fastest sorting algorithms out there. This will make the big O notation to become O(n) and it will be the best case since the sorting algorithm will only have to loop once to check that all the elements have been sorted in the proper order.

The cases:

Worst/average case: the Bubble Sort algorithm's inner loop executes n times n/2 times (n*[n/2] =n^2/2) in the worst and average cases, which is where the big O notation of O(n^2) originates from. The worst case of Bubble Sort occurs when the smallest element is at the very end of the array. This generates a worst case since every time the outer loop loops, the largest misplaced element is put in the correct order as in each big element bubbles up until the smallest element becomes the largest misplaced element and gets sorted. This requires the algorithm to loop n times of n/2 to have the array fully sorted.

Best case: the best case occurs when the dataset is already fully sorted so the algorithm is only required to loop through the array once to confirm the each element is where it is supposed to be and it will exit the loop. This makes the algorithm only loop one time creating a big O notation of O(n) which is the best case.

Example of Bubble Sort run:

First loop

{7 1 4 2 19} → {3 7 1 2 19} //7 and 1 are compared and 1 is smaller than 7 so the two are swapped.

{1 7 6 2 19} → {1 6 7 2 19} //7>6 swap the two elements

{1 6 7 2 19} → {1 6 2 7 19} //7>2 swap the two elements

{1 6 2 7 19} →{1 6 2 7 19}//19>7 no need for swap

Second loop

{1 6 2 7 19} →{1 6 2 7 19} //1<6 no need swap correct order

{1 6 2 7 19} → {1 2 6 7 19}//1>2 swap two elements

{1 2 6 7 19} → {1 2 6 7 19}// 7>6 no need swap correct order

{1 2 6 7 19} →{1 2 6 7 19}//7<19 no need swap correct order

Even though array can be visually seen to be solved, algorithm needs to run one more time to because it does not recognize it until it performs it check to see if all the elements are in the right spot.

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third loop

{1 2 6 7 19} →{1 2 6 7 19}

{1 2 6 7 19} →{1 2 6 7 19}

{1 2 6 7 19} →{1 2 6 7 19}

{1 2 6 7 19} →{1 2 6 7 19}

Quicksort:

Quicksort is one of the fastest sorting algorithms to use and it can be proven with the graph and table above presenting its blazing fast sorting speeds.

How Quicksort works:

Quicksort works by choosing any index in the array and calling it the "pivot" point (in the code I handed in I made the "pivot" the last value of the array. After the pivot is established, all the elements are partitioned as in the elements less than the pivot are moved to the left and all the elements greater than the pivot are moved to the right. In more detail, there is a recursive call that sorts everything on the right side and there's continuous right recursive calls until there are only 2 elements and then a left recursive call is called to sort the left part of the 2 elements and then it goes back to the previous right recursive call and then it calls a left recursive call and this process continues until all elements less than the pivot are on the left and all elements greater than the pivot on the right. If the list is sorted, then the algorithm ends but if it isn't a new pivot is selected ant eh process occurs again until the whole array is fully sorted. Also, this all occurs inside the array just like how Selection Sort works.

Cases:

Worst case:

The worst case that can possibly occur with quick sort is when the pivot chosen is either the biggest or the smallest element in the whole array so all that's left after the partition is 0 on one side and n-1 on the other side. So the big O notation that represents this situation is $O(n^2)$

because it will have keep choosing pivots until the pivot is actually usable as in it won't provide the unbalanced situation with one side having 0 and the other side having n-1 elements.

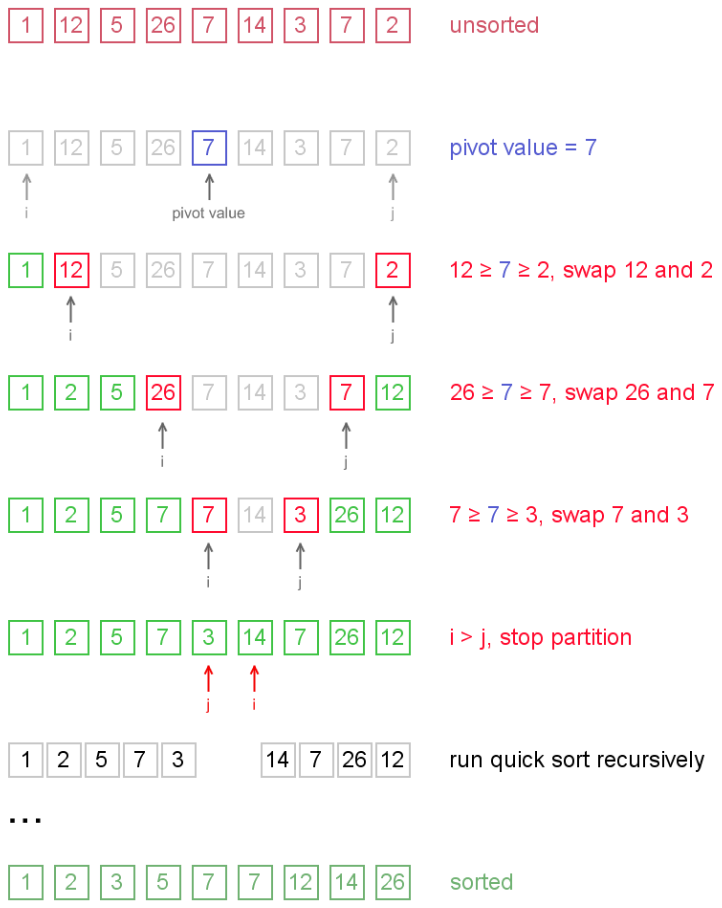
Average case:

An average case of quick sort has a big O notation of O(n log n) and occurs when the pivot point that is chosen provides a decent sort that is not 0 on one side and n-1 on the other. So, it will have a usable balance after the partition is complete.

Best:

The best case of a quick sort also has a big O notation of O(n log n), and this occurs when the pivot point chosen provides an almost each split of elements on the left and right side after the partition occurs. The process continues of choosing a pivot and partitioning and providing 2 nearly equal halves with every recursive call until the list is fully sorted.

Example of a Quick Sort run:



(Wikipedia)

<u>Merge Sort:</u>

Merge Sort is a sorting algorithm that can be described as a “divide and conquer” strategy where the algorithm breaks the unsorted array into unsorted subarrays. Then solutions for the sub problems are found and the solutions to the sub problems are used to solve the actual problem.

How Merge Sort works:

Merge Sort first starts off by checking if the final index is greater than the initial index. If it is the algorithm begins. First it finds the mid point of the array by taking the final index subtract one divided by two and adding the initial index. After that there is a recursive call that calls Merge Sort again but with values from the initial index to the mid point index and it keeps doing this until it is broken down into a sub array with 2 elements then it returns the array and calls another recursive Merge Sort call for the midpoint index plus one to the final index. It also performs the same steps by breaking down the algorithm into sub problems and then once sorted it returns. After each left and right subarray is sorted the merge function is called. The merge function first starts off by creating temporary arrays that will store the data from the left and right sub arrays. Then the merge function begins to combine the two sub arrays. This is done by checking if both sub arrays still have elements that can be combined, if this is true check to see which subarray has the smallest element and that will be the one put in order and then advance the index number of the subarray which it was taken from and advance the index of the array that will be the output. If one subarray runs out of elements to merge first, it will just add the elements from the sub array to the output array and advance the position of the subarray it is taken the element from as well as the output array.

Cases:

For the worst, average and best run times of Merge Sort, the typical Big O notation is O(n log n) where is in the number of elements that need to be sorted. The question is, where doe the Big O notation of O(n log n) come from?

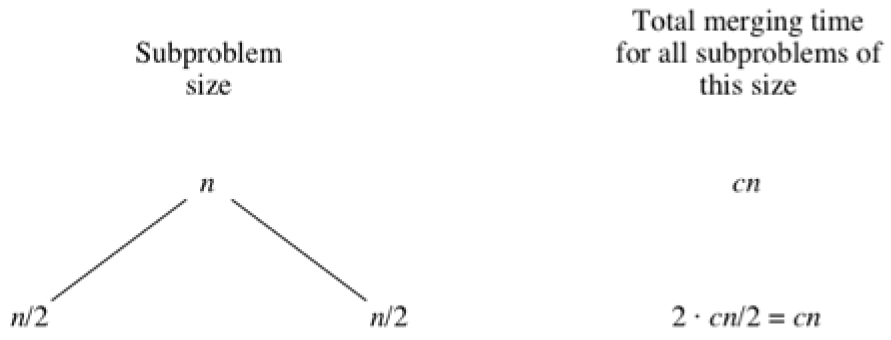
The first part is when the algorithm begins to divide the array and that takes constant time to perform and then it pin points a mid point in the array and then constant time is indicated by O(1).

The second step can be called the “conquer” step where the algorithm recursively sorts each of the sub arrays in the array and each sub array takes n/2 time to solve.

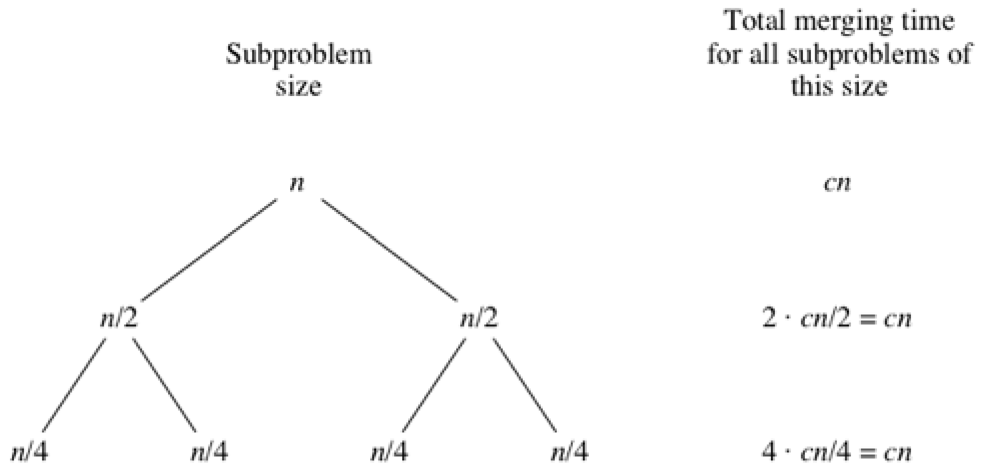
Then the third and final step is combining the two arrays so it will take O(n) time to merge the two sorted subarrays with one another. The O(1) step can be called a constant such as *C* and the divide and combine step can be simplified to be *c*n time to complete.

An assumption can be made that n is always greater than 1 so when n/2 is dealt with it is always a whole number. Another thing is that since there are two sub arrays the running time of Merge Sort can be thought of as twice n/2 times which is n/4 times for the running time to call the two sub arrays then it would take *c*n/2 time to merge one array and since there are two arrays it would be 2*(*c*n/2) which simplifies to the original sort and merge equation above of *c*n.

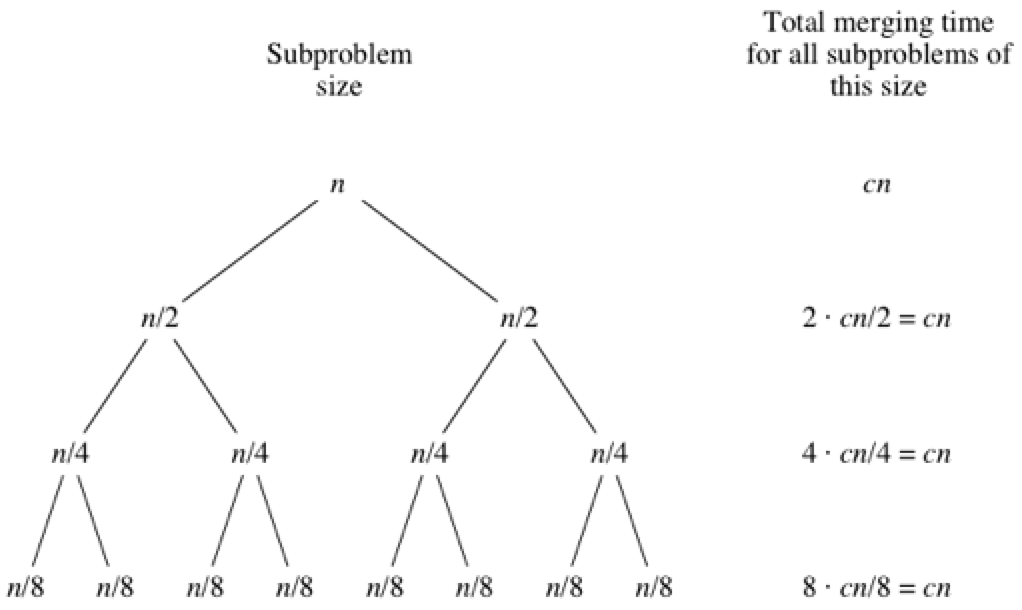
A merging example and showing where O(n log n) comes from:



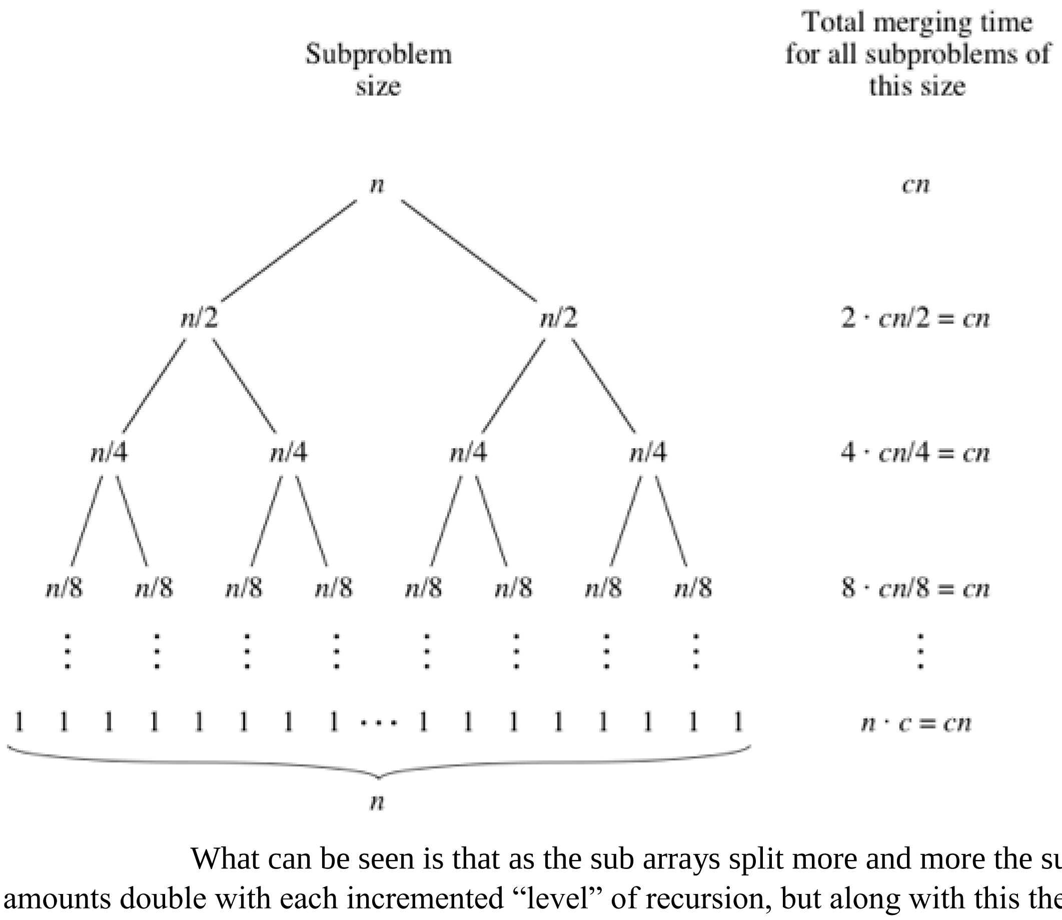
I   n the tree graph about it can be seen that n is the original array or known as the root node and then the two n/2 are the child nodes since they are the two sub arrays from the original array.



Then each of the two sub arrays recursively sorts two more sub arrays which will take (n/2)/2 time or n/4 and then the two sub arrays will become four sub arrays and each of the four sub arrays merge n/4 elements each so the merging time would be *c*n/4 time to fully merge. The total merging time would be 4*(*c*n/4) which simplifies to *c*n.



Then if the other four subarrays each split into two more sub arrays each then the total size of the sub arrays becomes n/8 per sub array and the merging time would be *c*n/8 per sub array and a total of 8*(*c*n/8) which simplifies to *c*n still.



What can be seen is that as the sub arrays split more and more the sub array amounts double with each incremented "level" of recursion, but along with this the time to merge halves. In turn the doubling and merging cancels out one another and leaving a total merging time of *c*n for each merged level of recursion.

Throughout the sorting process the array will eventually reach the point where the sub arrays will have a size of 1 which is the base case and it takes O(1) to sort that sub array of size 1. This is because there is a necessary check to see if the initial array index is less than the end array index.

Along with this, in the beginning there were n elements in the array so there are also n number of sub arrays with a size of 1 and each of these sub arrays take O(1) time to sort the total sorting time of all the base cases of the array will take *c*n time.

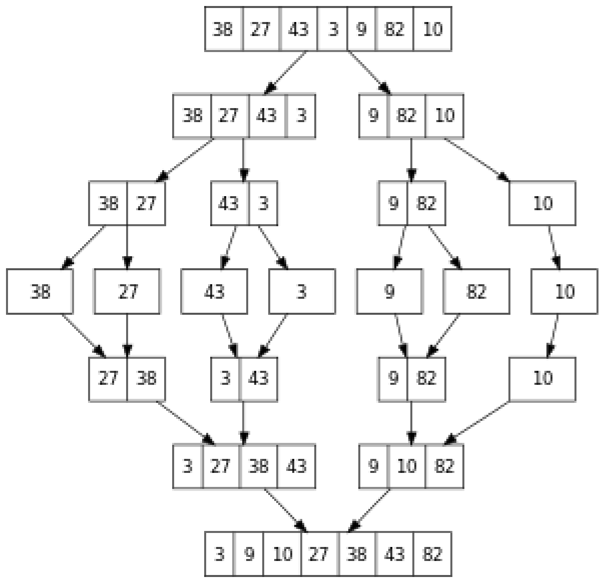
Now that the understanding of how the sub arrays are attained is established, a variable that can be used to represent the total time to merge all the levels in the tree graph is *l*. The total merging time of all levels would be *l**(*c*n). The process basically starts with n amount of elements that are continuously halved into sub arrays until the subarrays have a size of 1 so the equation would look something like *l* =log n +1.

An example would be if there were an array with an n value of n =8. Then the equation would become log n +1 =4 and this array would have a tree with 4 levels which would be 8, 4, 2, and 1 and the total time to perform this merge sort would be *c*n(log n +1). Since in Big O notation all that is cared about is the largest order term in a polynomial equation, the +1 becomes negligible and *c* is just a constant of 1. So this is where the O(n log n) big O notation comes from for the algorithm.

Other facts about Merge Sort:

Merge Sort is a sorting algorithm that has to copy data from the original array into two temporary sub arrays which can be considered the lower half and upper half of the array. Since Merge Sort does not perform all of its sorting in the original array it can be say that Merge Sort is not a work in place sorting algorithm. Why does it matter if Merge Sort is not a work in place sorting algorithm? In comparison to work in place algorithms such as Selection Sort and Insertion Sort all their sorting is done in the array, and thus uses less memory to sort the array. This is only an issue if memory is limited and the preferred sorting algorithm will be an in-place sorting algorithm to reduce memory usage.

Example of a Merge Sort run:



Gnome Sort:

Gnome Sort is a very straight forward algorithm in that it is easier to code than Bubble Sort.

How Gnome Sort works:

Gnome sort works by setting the initial index to 0. If the current index is 0 or the value at the current index is less than the value of the adjacent index, the current index number advances. If the value at the current index is greater than the value at the adjacent index, swap the two values. Then the current index moves back one and the subtracted index becomes the current index. The value at the current index is checked again and if it is greater than the value at the adjacent index, the two values are swapped and the current index moves back by 1 again, but if the value at the current index is less than the adjacent index the current index advances. This processes occurs until the array is fully sorted.

Cases:

Worst case:

Gnome Sort has a big O notation of O(n^2) for its worst case. This occurs when the smallest element or the array is arranged from left to right from greatest to bigger since the current index will have to keep advancing and swapping and de-advancing swapping and advancing again until the array is fully sorted.

Average case:

Gnome Sort has a big O notation of O(n^2) for its average case. This occurs when the array is arranged in a way where there is a requirement for advancing swapping, de-advancing, swapping, de-advancing, swapping and advancing again.

Best case:

Gnome Sort has a big O notation of O(n) for its best case. This occurs when there is no need to de-advance at all since the array is already fully sorted so the array can just keep advancing until it reaches the end of the array.

Current array	**Action to take**
[5, **3**, 2, 4]	a[pos] < a[pos-1], swap:
[3, **5**, 2, 4]	a[pos] >= a[pos-1], increment pos:
[3, 5, **2**, 4]	a[pos] < a[pos-1], swap and pos > 1, decrement pos:
[3, **2**, 5, 4]	a[pos] < a[pos-1], swap and pos <= 1, increment pos:
[2, 3, **5**, 4]	a[pos] >= a[pos-1], increment pos:
[2, 3, 5, **4**]	a[pos] < a[pos-1], swap and pos > 1, decrement pos:
[2, 3, **4**, 5]	a[pos] >= a[pos-1], increment pos:
[2, 3, 4, **5**]	a[pos] >= a[pos-1], increment pos:
[2, 3, 4, 5]	pos == length(a), finished.

(Wikipedia)

Bibliography

Xoaxdotnet. "Algorithms Lesson 7: Analyzing Bubblesort." YouTube. April 14, 2011. Accessed March 22, 2017. https://www.youtube.com/watch?v=ni_zk257Nqo.

Mycodeschool. "Analysis of Merge sort algorithm." YouTube. July 06, 2013. Accessed March 22, 2017. https://www.youtube.com/watch?v=0nlPxaC2lTw&t=7s.

"Bubble sort." Wikipedia. March 19, 2017. Accessed March 22, 2017. https://en.wikipedia.org/wiki/Bubble_sort.

"Gnome sort." Wikipedia. March 22, 2017. Accessed March 22, 2017. https://en.wikipedia.org/wiki/Gnome_sort.

"Gnome Sort - The Simplest Sort Algorithm." Gnome Sort - The Simplest Sort Algorithm. Accessed March 22, 2017. https://dickgrune.com/Programs/gnomesort.html.

"Khan Academy." Khan Academy. Accessed March 22, 2017. https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/analysis-of-merge-sort.

"Khan Academy." Khan Academy. Accessed March 22, 2017. https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/overview-of-merge-sort.

"Merge sort." Wikipedia. March 21, 2017. Accessed March 22, 2017. https://en.wikipedia.org/wiki/Merge_sort.

"Quicksort." Wikipedia. March 15, 2017. Accessed March 22, 2017. https://en.wikipedia.org/wiki/Quicksort#Worst-case_analysis.

"Selection sort." Wikipedia. March 21, 2017. Accessed March 22, 2017. https://en.wikipedia.org/wiki/Selection_sort.

GameConstructor. "Selection Sort - Algorithm and Analysis." YouTube. November 05, 2015.

Accessed March 22, 2017. https://www.youtube.com/watch?v=YbdzM0e8S5I.

Mycodeschool. "Selection sort algorithm." YouTube. June 07, 2013. Accessed March 22, 2017. https://www.youtube.com/watch?v=GUDLRan2DWM&list=PL2_aWCzGMAwKedT2KfDMB9YA5DgASZb3U&index=2.