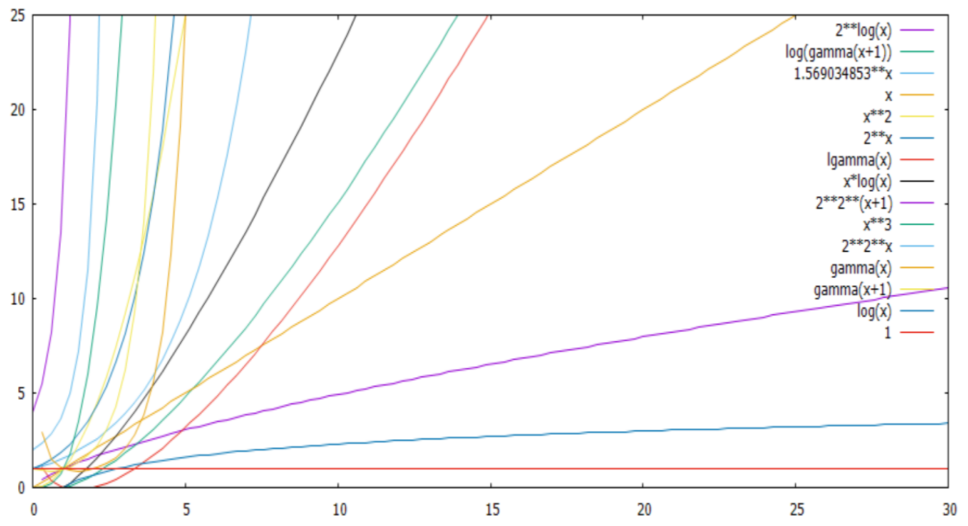
EX-1:

Function Rank Table (Highest to lowest)		
	$f(n)$	**(Bounds)**
1	$2^{2\wedge(n+1)}$	$\Omega\ (2^{2\wedge n})$
2	$2^{2\wedge n}$	$\Omega\ ((n+1)!)$
3	$(n+1)!$	$\Omega\ (n!)$
4	$n!$	$\Omega\ (2^n)$
5	2^n	$\Omega\ (e^n)$
6	e^n	$\Omega\ n^3$
7	n^3	$\Omega\ (n^2)$
8	n^2	$\Omega\ n\lg(n)$
9	$n\lg(n)$	$\Omega\ \lg(n!)$
10	$\lg(n!)$	$\Omega\ \ln(n)$
11	$\ln(n)$	$\theta(n)$
12	n	$\Omega\ (2^{\lg^* n})$
13	$2^{\lg^* n}$	$\Omega(\lg(n))$
14	$\lg(n)$	$\theta(1)$
15	1	

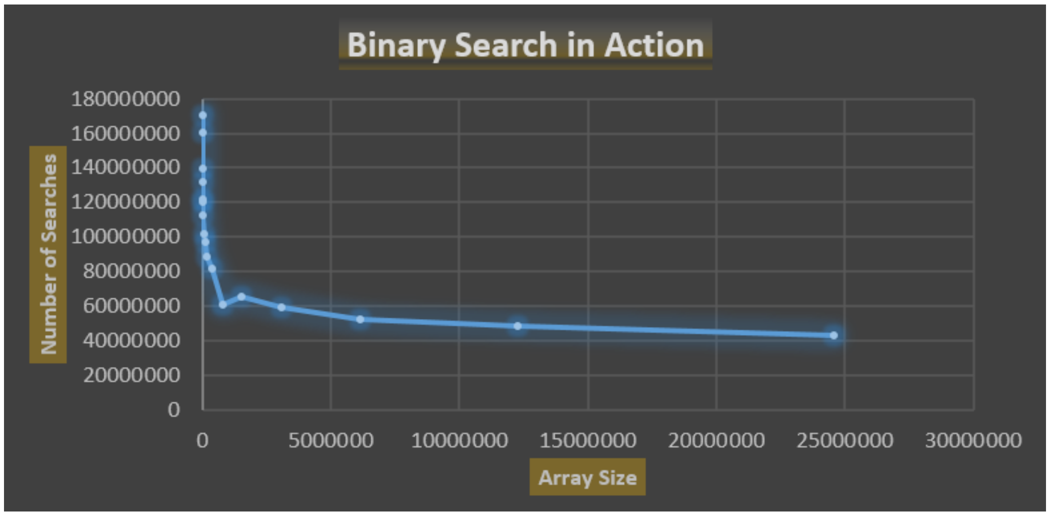


EX-2:

The following binary search is implemented using while loop it takes an already sorted array reference and an integer searching element. This method is called multiple times in a minute and every search count is recorded. This one-minute cycle is tested on arrays with different sizes. I started from relatively small array size i.e. 400 and increased size by 2 every time. The graph shows sudden drop initially but stabilizes after constant increase in array size. The analysis and code structure is as follows:

```
/**
* Binary Search Method Takes a sorted array reference
* and element which is to be searched and returns true if found..
**/
bool B_Search(ref  Int64[]  A,  Int64  elem)  {
    Int64 lower_end = 0, mid, higher_end  =  A.Length  -  1;
    while (lower_end <= higher_end) {
        mid =  (lower_end  +  higher_end)  /  2;
        if (elem  <  A[mid])  {
            higher_end  =  mid  -  1;
        }
        else if (elem  >  A[mid])  {
            lower_end = mid  +  1;
        }
        else {  // Console.WriteLine("Element found...");

            return true;
        }
    }  // Console.WriteLine("No Such Element");
    return false;
}
```



EX-3:

Given,

$$T(n) = 7T(n/3) + n^2$$

Here,

$$a = 7, b = 3$$

$f(n) = n^2$

$n^{\boldsymbol{log\ a\ base\ b}} = n^2$

$n^{\boldsymbol{log\ 7\ base\ 3}} = n^2 \quad => \quad n^{\mathbf{1.77+€}} = n^2$

$where\ € = 0.23$

$This\ is\ case\ 3\ of\ the\ Master\ Theorem\ i.e\ n^{\boldsymbol{log\ a\ base\ b} + €}$

$Now,$

$The\ regularity\ condition\ must\ be\ satisfied.$

$a\ .f\left(\frac{n}{b}\right) \leq c.f(n)$

$7.f\left(\frac{n}{3}\right)^2 \leq c.n^2$

$7\left(\frac{n}{3}\right)^2 \leq c.n^2$

$7\left(\frac{n^2}{9}\right) \leq c.n^2$

$\frac{7}{9} \leq c$

⇨ $c \geq \frac{7}{9}$

$Verification,$

$$7\left(\frac{n^2}{9}\right) \leq 7\left(\frac{n^2}{9}\right)$$

EX-4:

$As\ we\ are\ dividing\ into\ two\ halves\ and\ skipping\ one\ half\ the$

$Recurrence\ equation\ of\ Binary\ search\ is\ as\ follows,$

$$\boldsymbol{T(n) = T(n/2) + O(1)}$$

$Here,$

$$a = 1, b = 2$$

$$f(n) = O(1)$$

$$n^{\boldsymbol{log\ a\ base\ b}} = n^2$$

$$n^{\boldsymbol{log\ 1\ base\ 2}} = n^0 \qquad => \qquad n^{\mathbf{0}} = 1$$

$This\ is\ case\ 2\ of\ the\ Master\ Theorem\ i.e\ n^{\boldsymbol{log\ a\ base\ b}}.log^k n$

$So,$

$$T(n) = \theta(n^{\boldsymbol{log\ a\ base\ b}}.log^k n)$$

$$T(n) = \theta(n^{\mathbf{0}}.log\ n)$$

$$T(n) = \theta(log\ n)$$

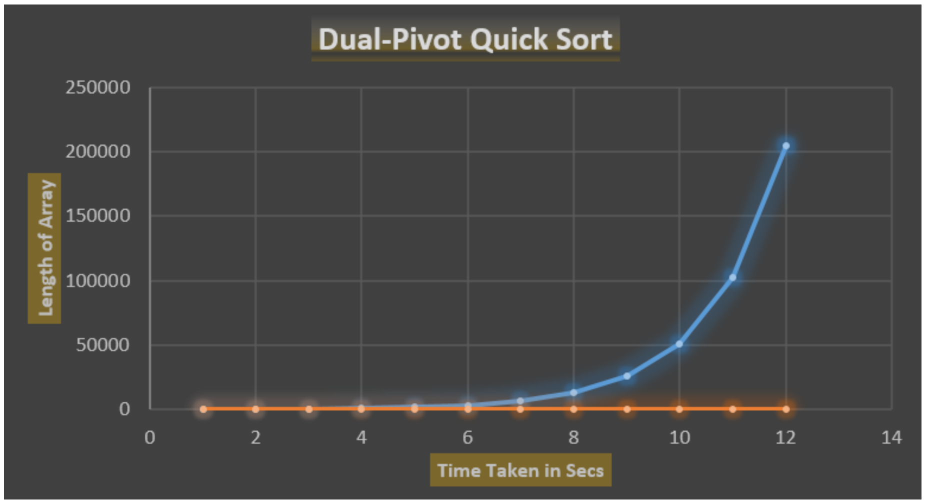
EX-5:

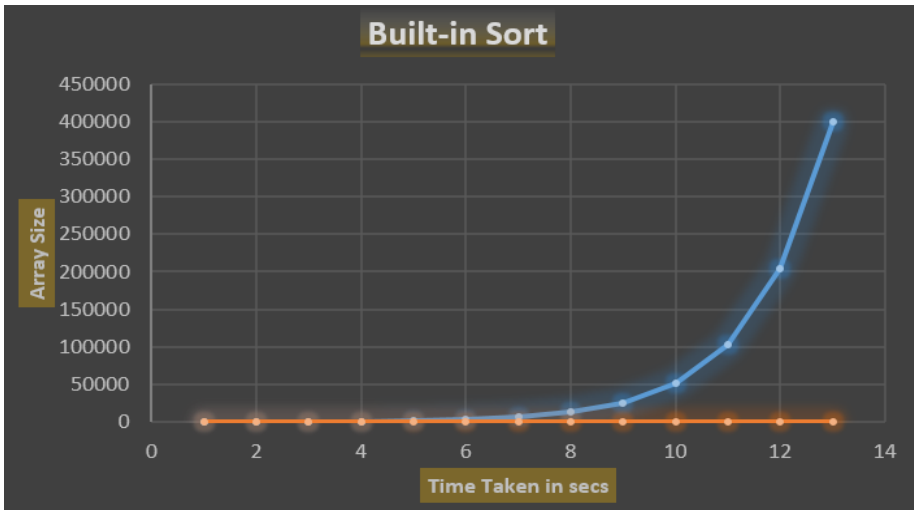
I implemented the following dual pivot quick sort by the help of this Java article. https://dzone.com/articles/algorithm-week-quicksort-three. The following technique is good but it had some extra calculation issues where it was comparing if pivot1==pivot2. I first tested it on java then implemented its variant in the language which I am using i.e. C#. So I tested this many times and calculated its running time per array size on different sized arrays.

The code is as follows:

```
        /**
          * DualPivot QuickSort..
          * Takes 3 arguments, The Array to be sorted 'A'[]
          * The lower boundery of the array 'lower'
          * The upper boundery of the array 'higher'
          **/
    void Dual_Pivot_QuickSort(int[] A, int lower, int higher) {
        if (higher <= lower) return;
        int p1 = A[lower];
        int p2 = A[higher];
        if (p1 > p2) {
            Swapping(A, lower, higher);
            p1 = A[lower];
            p2 = A[higher];
        }
        int i = lower + 1;
        int lt = lower + 1;
        int gt = higher - 1;
    while (i <= gt) {
        if (A[i] < p1) {
            Swapping(A, i++, lt++);
        } else if (p2 < A[i]) {
            Swapping(A, i, gt--);
        } else {
            i++;
        }
    }
    Swapping(A, lower, --lt);
    Swapping(A, higher, ++gt); // Recursion Split
    Dual_Pivot_QuickSort(A, lower, lt - 1);
    Dual_Pivot_QuickSort(A, lt + 1, gt - 1);
    Dual_Pivot_QuickSort(A, gt + 1, higher);
}
```

I compared this quick sort with built-in sort with different array sizes starting from 100 and then double the size each time. Following are the results of both sorting algorithms.





Its visible that when the input size grows the built-in sort beats the dual pivot sort. The reason is built-in sort is battle tested for different types of input sizes while dual pivot sort cannot be used as general sorting algorithm. Most of the programming languages use hybrid algorithms based upon the size of input array.

In case of quick sort, it is very important to choose pivots wisely because sometimes splits can divide array into semi-equal parts. The draw-back of semi-equal parts is that it increases running time so in worst case we might get unlucky and have $\mathbf{n^2}$ as recursion would only be comparing elements linearly. There are several techniques of choosing pivots and the famous one is 'median of three' but still there is a chance that the algorithm might lean towards bad splits. So, the algorithm should be designed in such a manner that it should handle bad splits effectively. As discussed earlier one approach which most of the built-in sorts use is shifting to insertion sort in such way even if it's a bad split the smaller size input will consume lesser time on insertion sort compared to recursive calls of quick-sort.