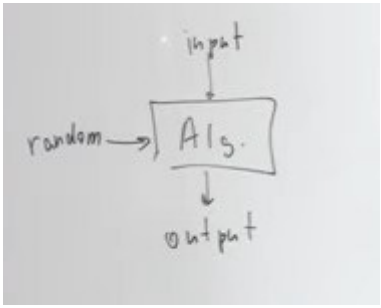
03. Quick Sort. Order Statistics

Quick Sort

Quick sort is a randomized algorithm.

Randomized Algorithms:

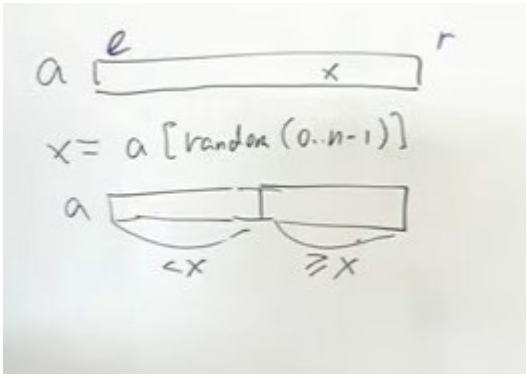
Usually, Input->[Algorithm]->Output happens
In randomized Algorithms:



Quick Sort:

1. Given an array, pick a random element x in the array.
2. Traverse the array and split(into two different arrays, not partitioning) it into two parts l and r. Keep elements <x in left part and >=x in the right part.
3. Recursively call the algorithm for both parts, picking x at random every time.

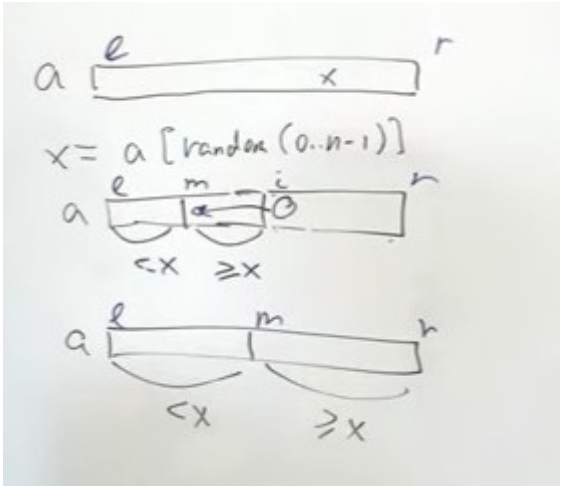
Improvement: Instead of making a new array at step 2, just modify the same array so all elements <x are in its left part and >=x are in its right part.



A segment is defined by its borders ('l'(index 0) and 'r'(outside index n-1) in the image)
Code:

```
sort(l, r)
    if r-l<=1 //Array having 0 or 1 element is already sorted
        return
```

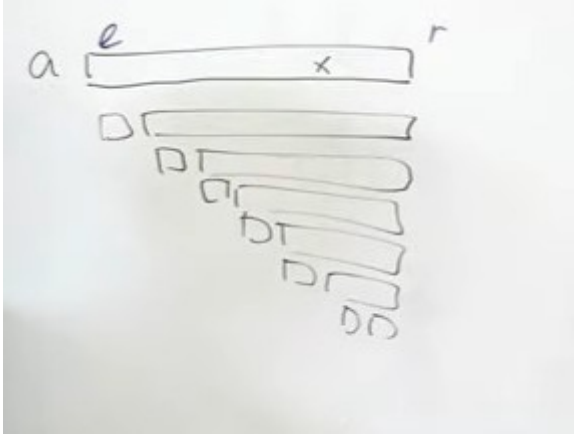
```
    x=a[random(l..r-1)] //Pick a random element
    m=l
            //Initially, splitting point is 0th index
            //Index < m will have elements <x
            //Index >=m will have elements >=x
    for i=l..r-1 //Traverse the array
        if a[i]<x //If we encounter an element < x
            swap(a[i],a[m]) //Take it to mth index
            m++ //Increase m to new split point
    sort(l,m)
    sort(m,r)
    //Recursively sort both parts
```



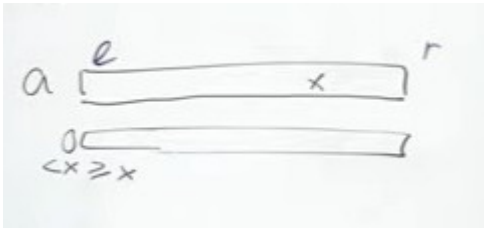
Above code is not the best implementation. **It doesn't always work.** If all elements are equal, no partitions will be made and recursive calls won't stop.

One way to solve this is to split the array in not two but three parts. <x, =x, and >x and recursively call the function for three parts.

Time complexity:
Worst case: Each time x is the maximum of minimum element in the array



O(n^2)
And in the given implementation it is even worse.



If x is minimum element everytime, the algorithm never stops.

This sounds like a big problem, but it's actually not. Since we pick the element randomly, the probability that we'll always pick the smallest element for infinite time is essentially zero.

Since this is a **randomized algorithm**, we won't measure the time complexity by looking at the worst case. Instead we'll consider the mean of number of operations.

$$E(T(n)) = \sum_{x} x \cdot p(T(n)=x)$$

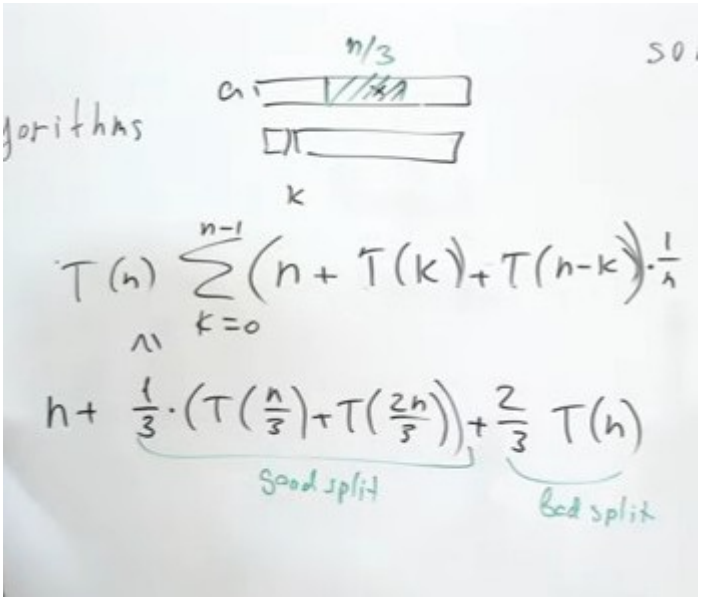
Here, T(n) is:

$$T(n) \sum_{k=0}^{n-1} \left(n + T(k) + T(n-k)\right) \cdot \frac{1}{n}$$

n to split the array. A recursive call of T(k) and another recursive call of T(n-k)
(Sum of all/n) is the mean value.

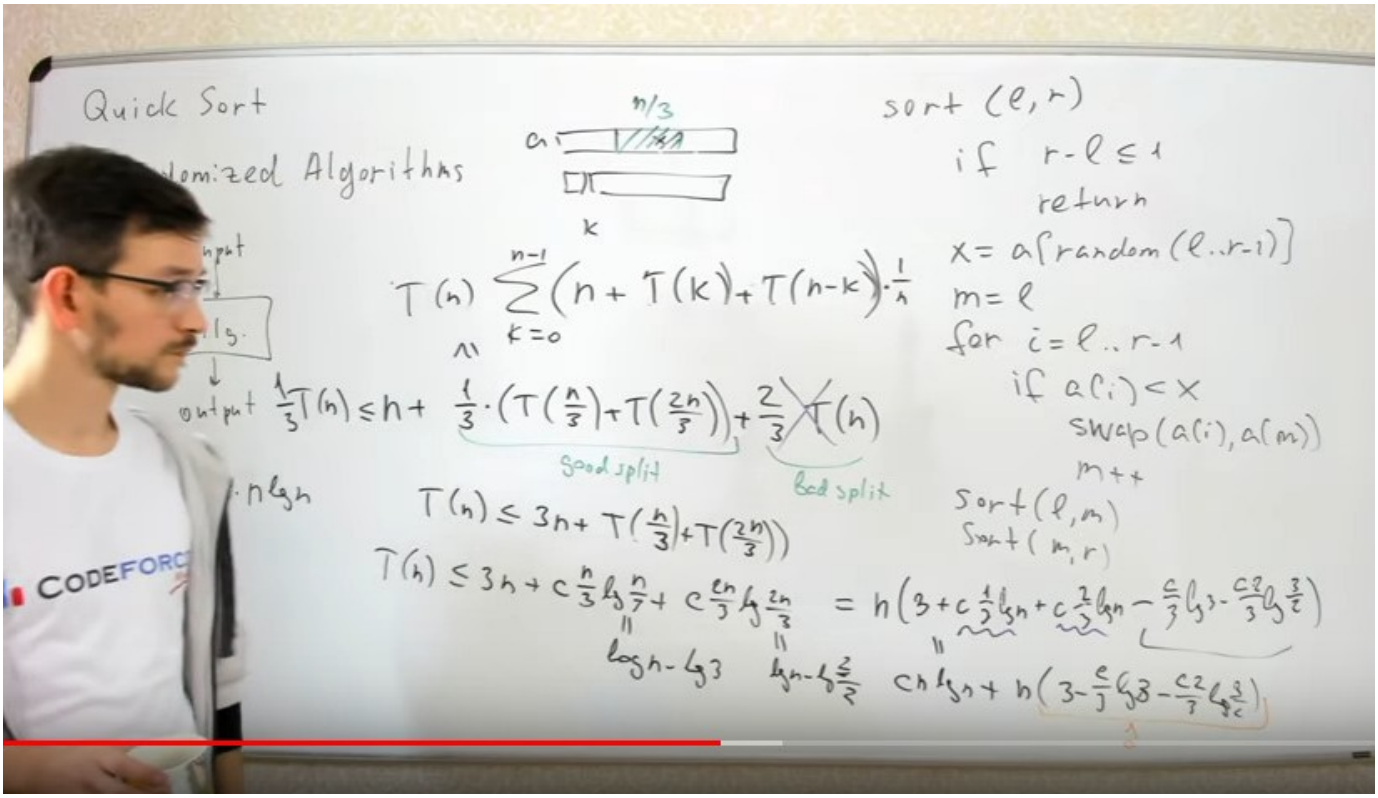
Now,
In the range of all the elements in the array, the first third and the last third sections have **bad choices**, as taking those elements causes an uneven split. The middle third of the array(shaded region) has **good choices**, as taking those elements splits the array fairly evenly.

Thus, we get the following inequality:



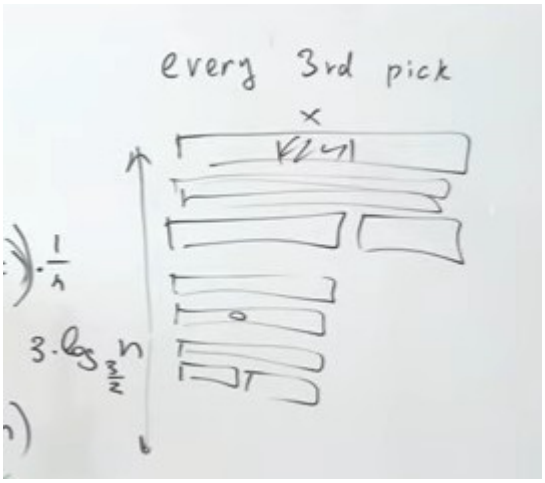
(1/3 * (n/3 and 2n/3 in good split) + 2/3 * (n for bad split))

By solving, we get this:



O(nlogn)

In essence, if roughly every 3rd pick is a good pick then that decreases the size of the recursion by some constant factor.



The depth of this recursion is of the order logn. Time for each is n. Thus, final time complexity is O(nlogn)

We can improve upon the constant factor in time complexity(not the asymptotic).

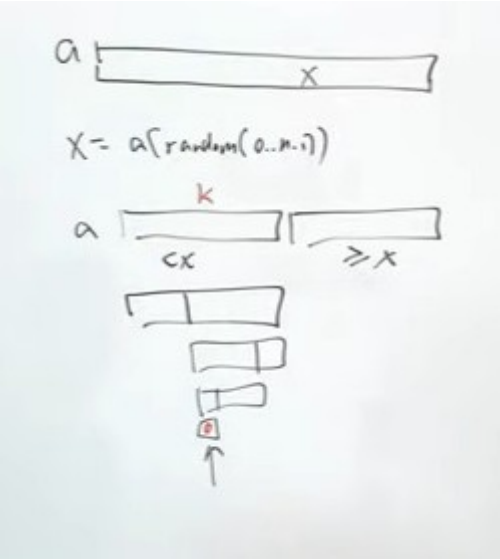
How to pick the elements from the good-pick (median) group with better probability?
Instead of one element, pick three or more elements, and out of those elements, pick the median as the choice.

Order Statistics

Q. Given an unsorted array a, and an index k, find a[k] in the sorted array.

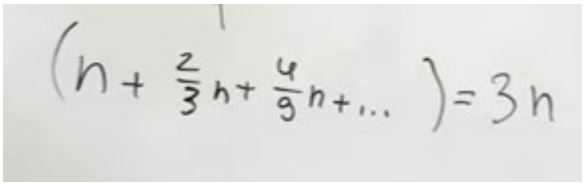
Sorting the array and finding the element will take nlogn time.
We can do it faster, without sorting the whole array.

1. Pick a random element x, and split the array into two parts like in quick sort.
2. Make a recursive call on only that part of the array which has the needed index




```
find(l, r, k) //l<=k<r
    if r-l=1 //If one element is left
        return a[k] //Answer is found
    x = a[random(l..r-1)] //Pick a random element
    m=l //Split point initially index 0
    for i=l..r-1 //Traverse the array
        if a[i]<x
        swap(a[i],a[x])
        m++
        //Make two parts just like in quicksort
    if k<m //If index k is in left half
        return find(l, m, k) //Recur for left half
    else
        return find(m, r, k) //Recur for right half
```

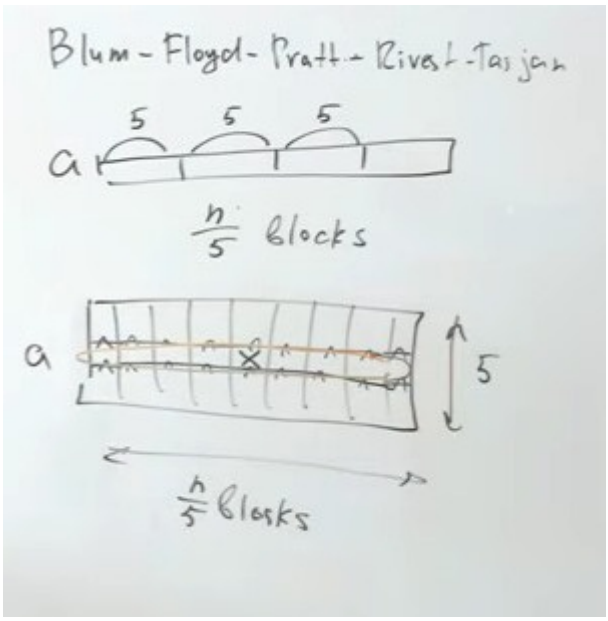
We only make one recursive call instead of two.
Assuming good splits, we get



This is average O(n)

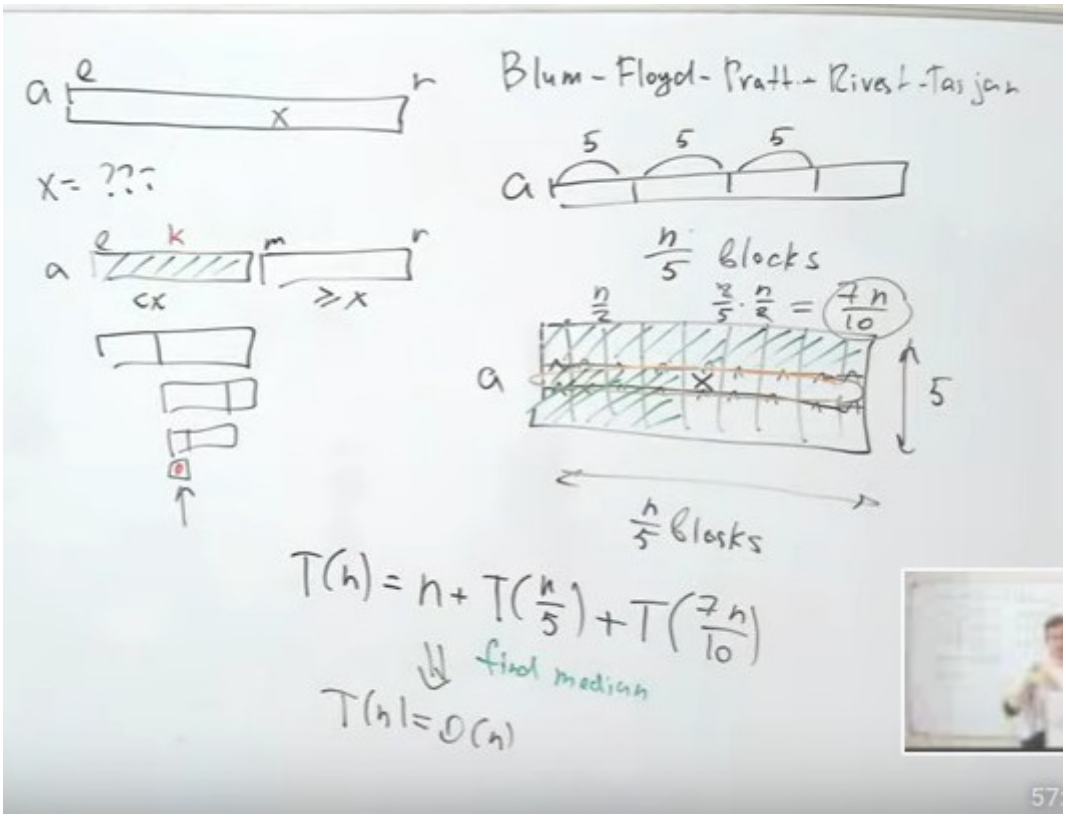
Blum-Floyd-Pratt-Rivest-Tarjan Algorithm

Used to find the element x, not in a random way, but in a determined way.



1. Partition the given array into blocks of size 5 (linear time).
2. Sort each block and find the median element in every block (linear time as size of block is small).
3. Find the median of all these calculated medians. This value x is a good pick.

NOTE: In step 3, it might appear that we need to sort to find the median a second time. But no! We can use the same algorithm recursively!



n to make groups and find the median of every group

T(n/5) to find the median from all groups using recursion.

T(7/10) as 7n/10 elements MAY be lesser than x. In the picture shaded region represents places where elements may be lesser than x.

Can be proved equal to O(n) using induction

Proof.