Implementation and analysis of randomized quicksort and tail recursive quicksort utilizing Lomuto partition and Hoare partition

-- K. Goutham kumar reddy

Analysis on

Randomized quicksort with Lomoto partition

Randomized quicksort with Hoare Partitioning

Tail recursive quicksort using the Lomuto partition

Tail recursive quicksort with Hoare Partitioning

Randomized quicksort Vs Tail recursive quicksort

Random element is chosen as pivot	Typically the first or last element is chosen as pivot
Helps avoid worst case by dividing the array in at least 1:99 ratio	No guarantee of 1:99 ratio to enter O(nlogn)
Time complexity is optimized	Time is not optimized
When the input size is really large, stack overflow may happen.	Tail recursion optimizes the space complexity of the function by reducing the number of stack frames in the memory stack.
Works with Lomuto Partition Hoare's Partition	Works with Lomuto Partition Hoare's Partition

Lomoto vs Hoare's Partition Schema

Usually, the last piece is selected as the pivot.	In order to avoid an infinite loop, the last element is never chosen
Two pointers are initialized at the same location of the array	It initializes two points at opposing ends of the array
More number of swaps	Less number of swaps
No effect on same elements in the array	Works best when the chosen pivot is mostly the median that is if the array is made of same elements or almost same elements O(nlogn)
No effect of sorted array on the method O(n^2)	Performs its worst when the data is already sorted O(n^2)

Pseudocode : quicksort

```
quicksort(arr[], lo, hi)
    if lo < hi
        p = partitionRandom(arr, lo, hi)
        quicksort(arr, lo , p-1)
        quicksort(arr, p+1, hi)
```

Start the quick sort and run it till we just have to sort 0 elements left to sort

Call random pivot selection if

Pseudocode : Randomized quicksort

```
partitionRandom(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)
```

A random number is chosen from the data
And swapped with last element for Lomuto partition and it would have swapped with first if it is Hoare Partition

Pseudocode : Randomized quicksort

```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo     // place for swapping
    for j := lo to hi – 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i
```

Lomuto partition procedure is started
With 2 pointers starting at the start of the array a
And when j encounters element lesser than pivot
arr[i] is swapped with arr[j]
When j reaches the end of array arr[i] is swapped with arr[hi]

Hoare's partition

```
def partition(arr,start,stop):
 pivot = start # pivot
    i = start - 1
    j = stop + 1
    while True:
        while True:
            i = i + 1
            if arr[i] >= arr[pivot]:
                break
        while True:
            j = j - 1
            if arr[j] <= arr[pivot]:
                break
        if i >= j:
            return j
        arr[i] , arr[j] = arr[j] , arr[i]
```

Hoare's partition procedure is started
With 2 pointers one starting at the start of the array and the other at the end.
Keep incrementing i till we find a element greater than i
Keep incrementing j till we find element smaller than j
If i is greater than j return j and swap the j th element with i

Tail Recursion

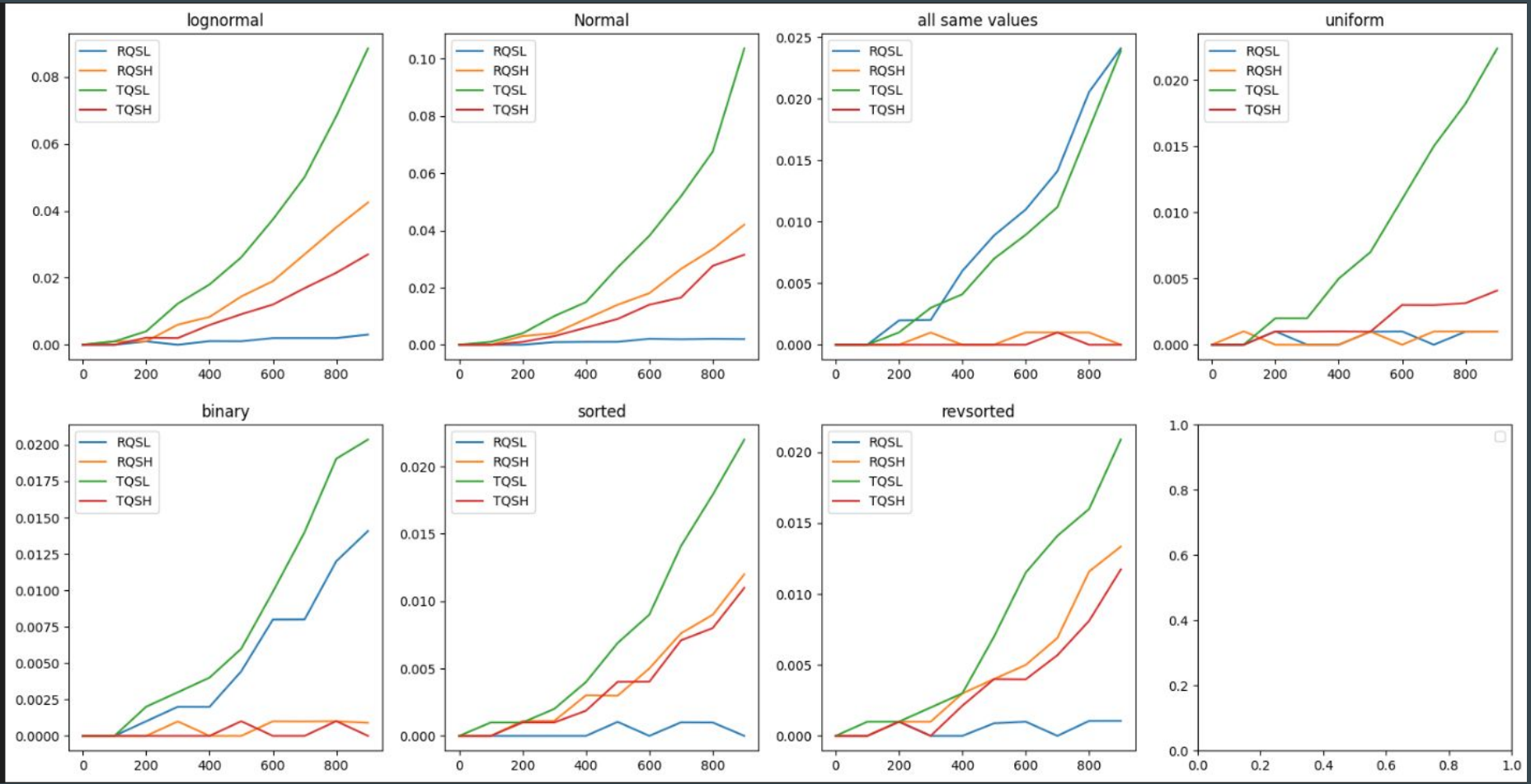
```
def quicksort(a, start, stop):

    while (start < stop):
        pivot = partition(a, start, stop);

        # If left part is smaller, then recur for left
        # part and handle right part iteratively
        if (pivot - start < stop - pivot):
            quicksort(a, start, pivot - 1);
            start = pivot + 1;

        # Else recur for right part
        else:
            quicksort(a, pivot + 1, stop);
            stop = pivot - 1;
    # print(a)
```

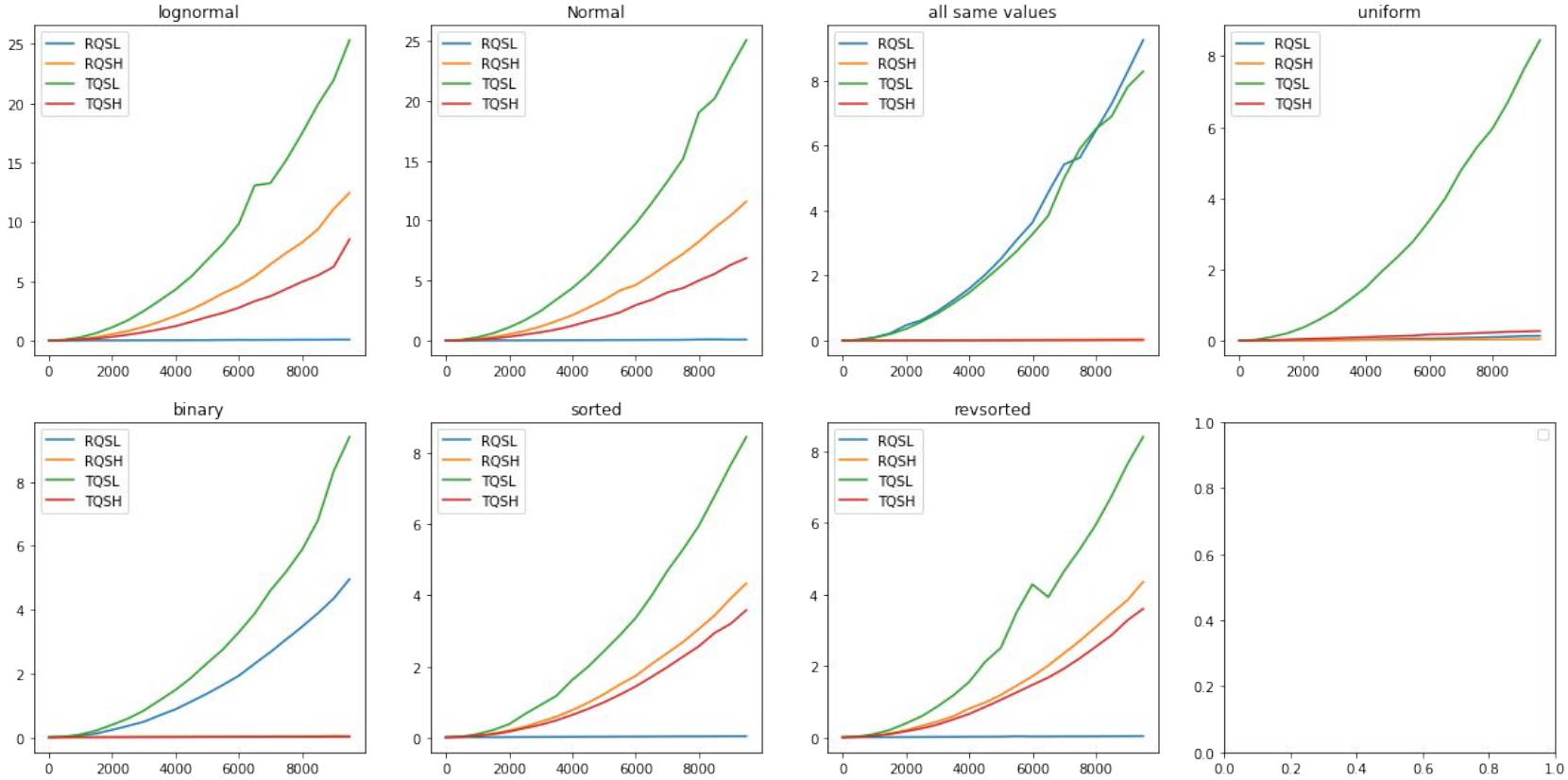
Start the quick sort and run it till we just have to sort 0 elements left to sort

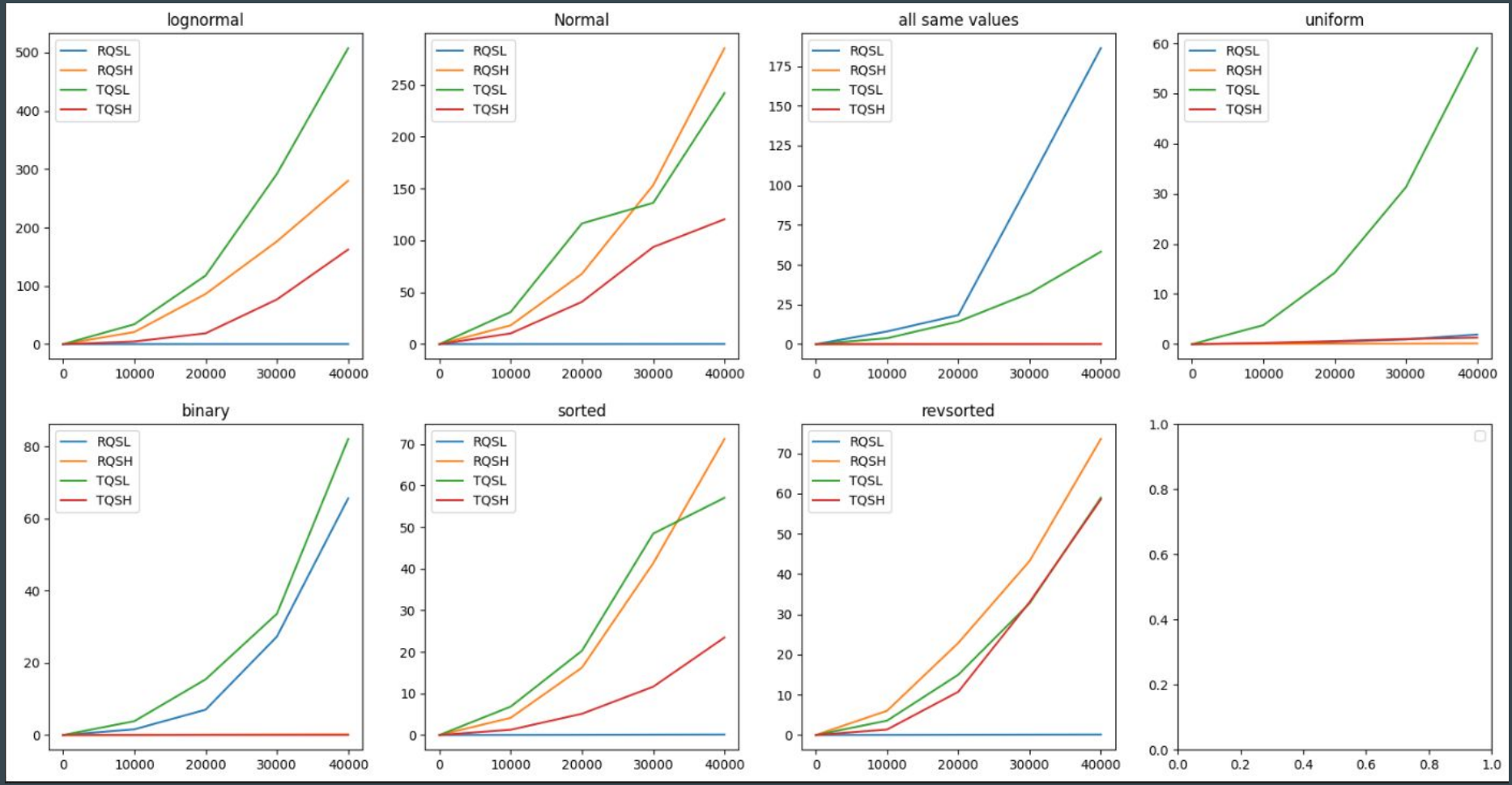
Unlike normal quicksort we check if left part is smaller, then recur for left part and handle right iteratively Else do the opposite

Data used for Analysis

1. Sorted
2. reverse sorted
3. All same values
4. 50% same values (binary)
5. Random numbers (uniform distribution)
6. Normal distribution
7. log normal distributions


lognormal
Normal
all same values
uniform
binary
sorted
revsorted
RQSL
RQSH
TQSL
TQSH


lognormal
Normal
all same values
uniform
binary
sorted
revsorted
RQSL
RQSH
TQSL
TQSH


lognormal
Normal
all same values
uniform
binary
sorted
revsorted
RQSL
RQSH
TQSL
TQSH

Analysis : Time complexity

Normal / Lognormal : RQSL < TQSH < RQSH < TQSL

Binary/unary: RQSH < TQSH < TQSL < RQSL

Sorted/revsorted : RQSL < TQSH < RQSH < TQSL

Uniform/random : RQSH <= RQSH <= TQSH <=TQSL

Runtime-analysis

Using random pivoting we improve the expected or average time complexity to O (N log N). The Worst-Case complexity is still O (N^2).

Using Tail recursion we improve the space complexity. The Worst-Case time complexity is still O (N^2).

Using Hoare's partition we improve the time complexity of unary or binary data to a great extent and worsens when data is sorted but in an average case it is better than lomotu.

	RQSL	TQSL	RQSH	TQSH
Normal/lognormal	O(nlogn)	O(n^2)	O(n^2)	O(n^2)
Sorted/revsorted	O(nlogn)	O(n^2)	O(n^2)	O(n^2)
unary/binary	O(n^2)	O(n^2)	O(nlogn)	O(nlogn)
uniform/random	O(nlogn)	O(n^2)	O(nlogn)	O(nlogn)

Best time applications

Normal/Lognorm : Lomotu or hoare partition with Randomized pivot

Binary/Unary : HOARE's partition with tail recursion

Sorted/Revsorted : Lomoto with randomized pivot

Uniform/Random : Lomoto or hoare with randomized pivot

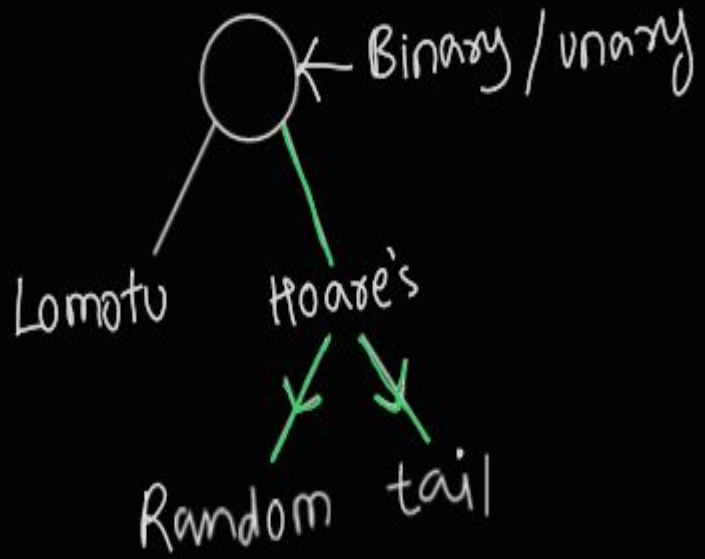
Conclusion

Random is better than tail recursion except when the data is binary or unary

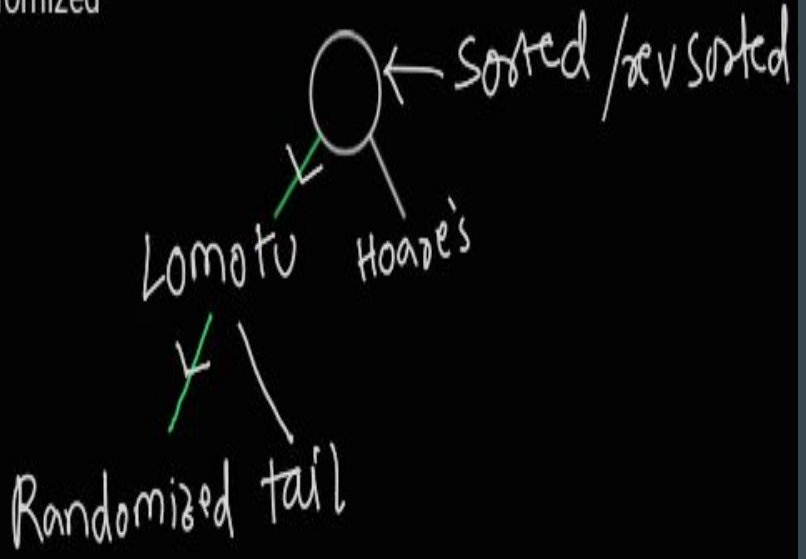
Hoare is better than lomotu except when data is sorted or reverse sorted

Decision Trees

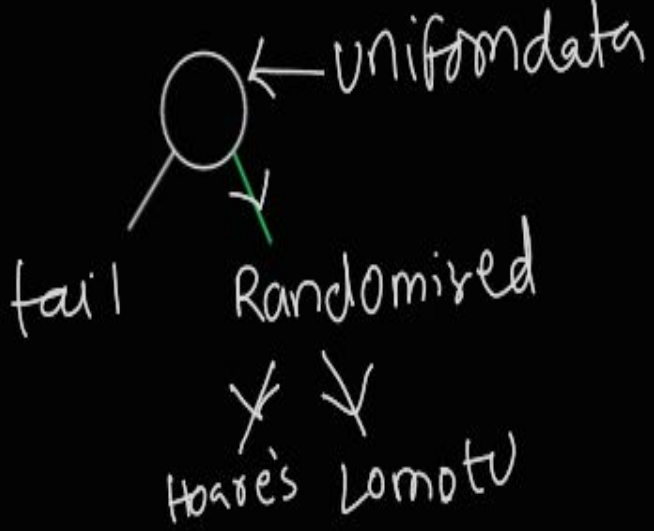
When data is binary or unary choose HOAR's partition



Sorted or reverse sorted input choose lomotu with randomized



Uniformly distributed data choose Randomized pivot



For lognormal and normal input type choose randomized with