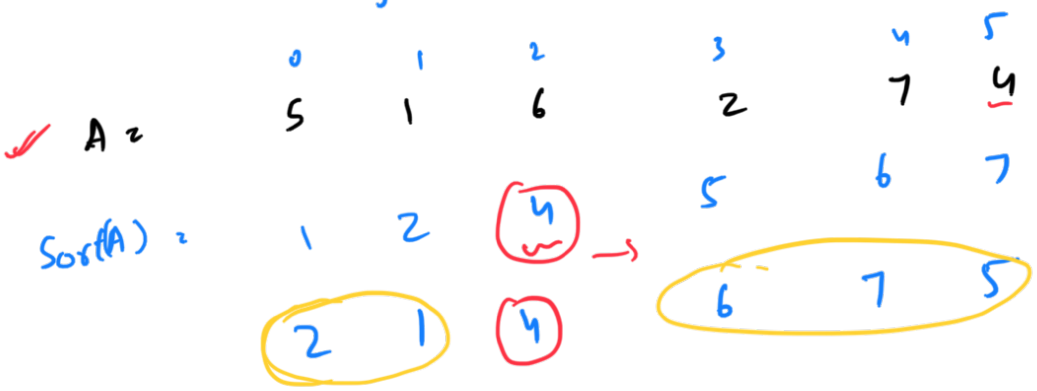
# Sorting - 2

1) Quick Sort
2) Bucket sort
3) Radix Sort
4) Problem on Bucketisation ✓

Question: Given an array of integers, rearrange the array such that the last element (A[n-1]) is in the correct position of a sorted array

-) All elements lesser than it should be in the left

-) All elements greater(equal) should be in the right

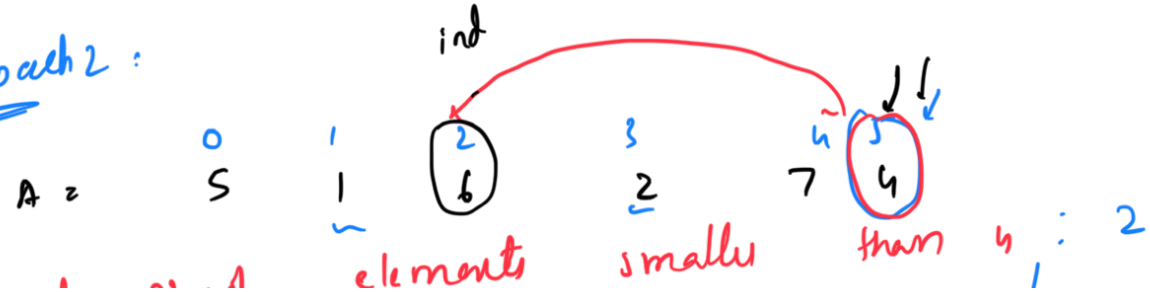



Approach 1: Sort the array

T.C: O(nlogn)

(K) ∈[0, N-1) arr[K]

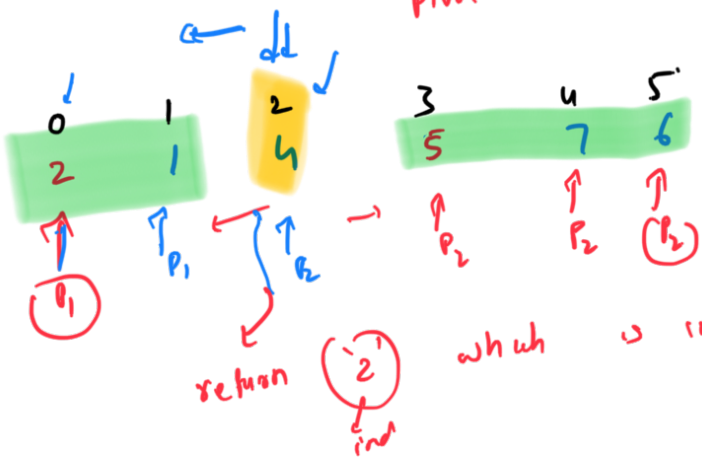
Approach 2:



→ find No. of

pivot = 4

O(n)




$P_1 = 0$, $P_2 = n-1$

which is index of last el

```
int partition ( A[], N) {
pivot = A[N-1];
for(int i=0; i<N; i++) {
    if( A[i] < pivot)
        ind++;
}
swap (A[ind], A[n-1] );
```

O(n) { (count of smaller elements)

$p1 = 0$, $p2 = n-1$;

O(n) {

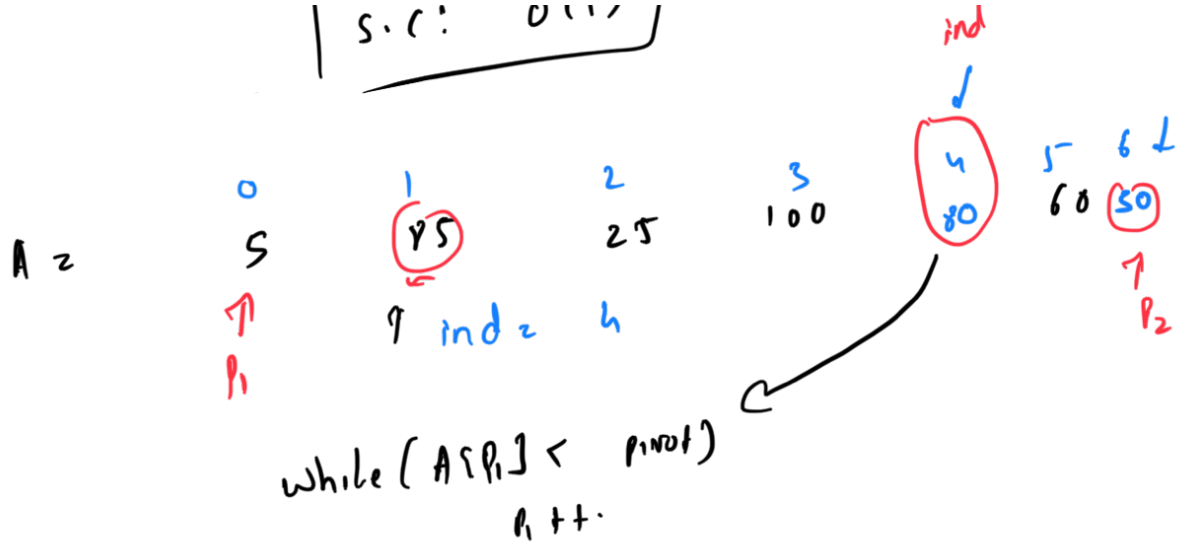
```
while ( p1 < ind && p2 > ind ) {
    while (A[p1] < pivot)
        p1++;
    while (A[p2] >= pivot)
        p2--;
    swap ( A[p1], A[p2] );
}
return ind;
```

(length check) (terminate) (Bring p1 to right position) $A[p_1] > A[p_2]$

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | $P_1$ | $P_1$ | ind | $P_2$ | $P_2$ | $P_2$ | $P_2$ | $P_2$ |
| A = | 2 | 1 | 4 | 4 | 4 | 4 | 6 | 7 |
| | | | | | | | 6 | 7 |

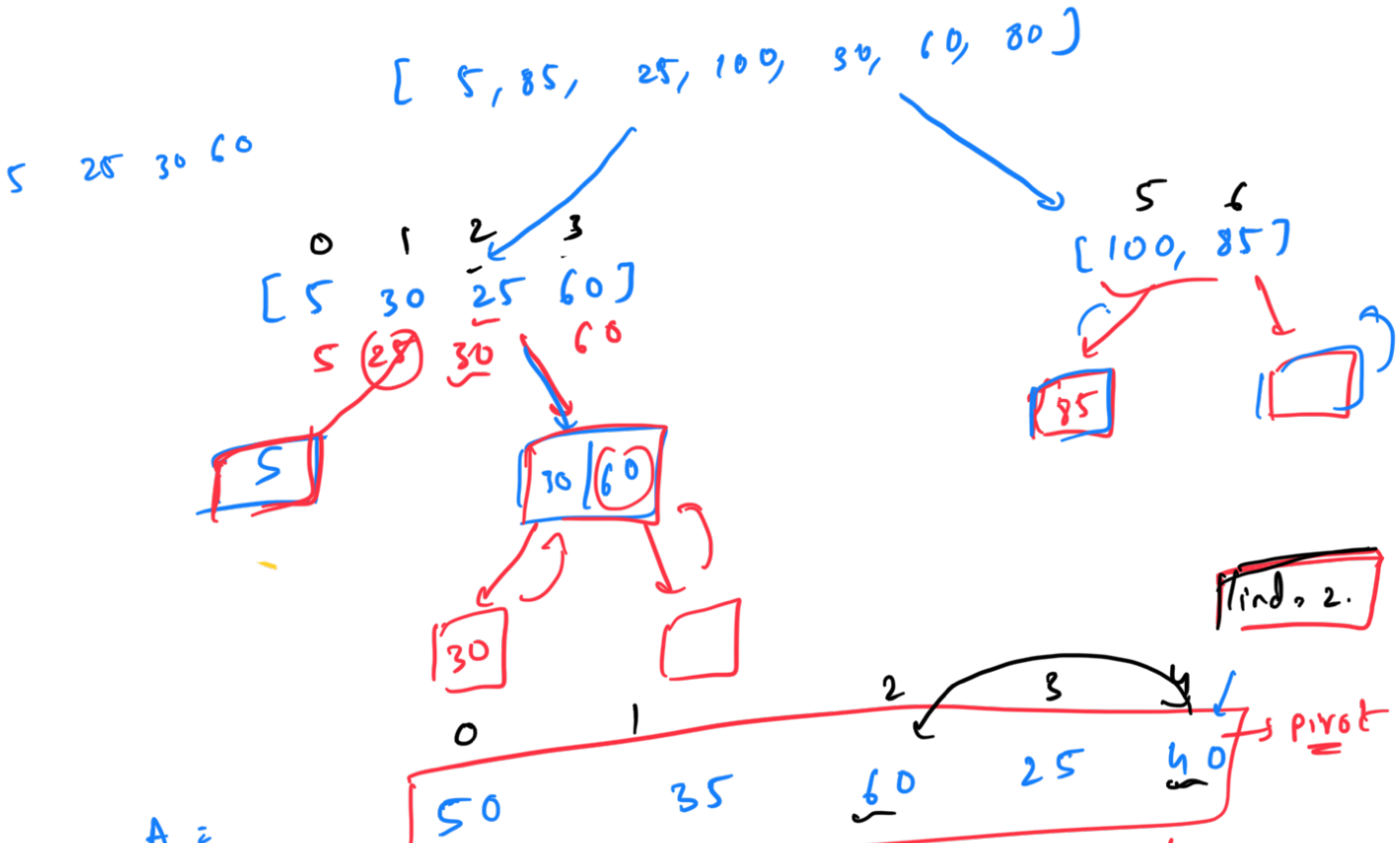
ind = 2 {1, 2}

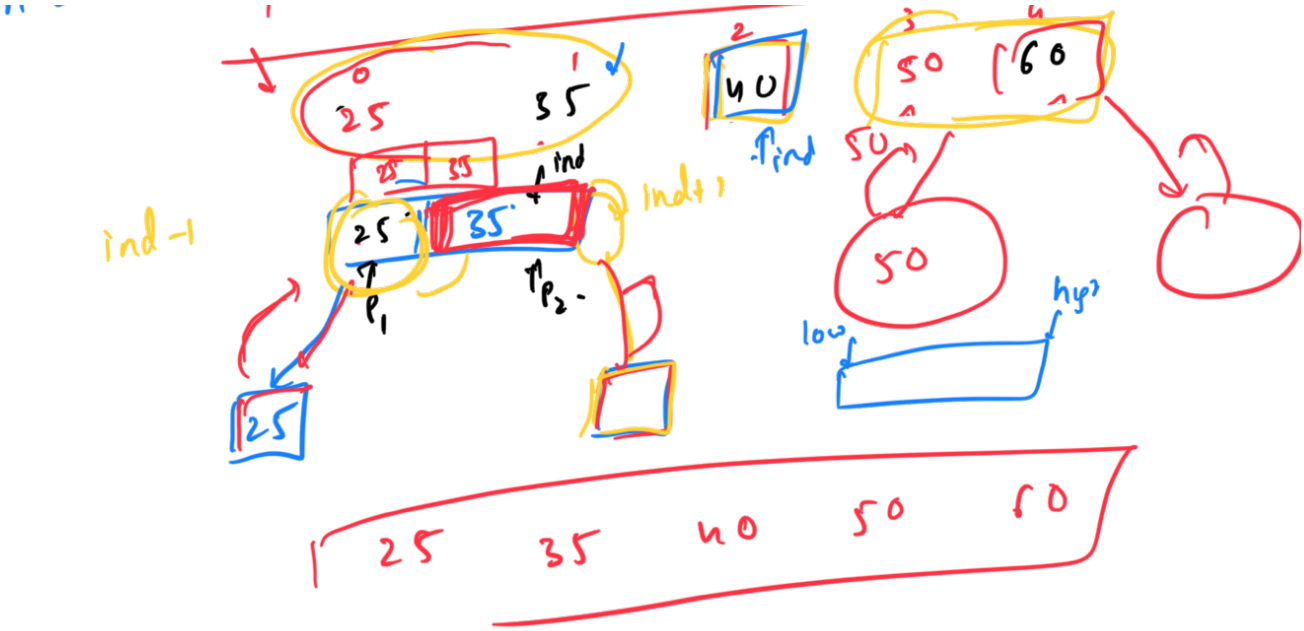
T.C: O(n)

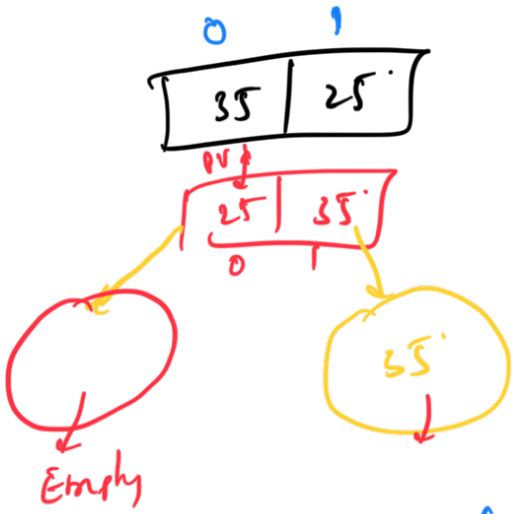



Quick sort:

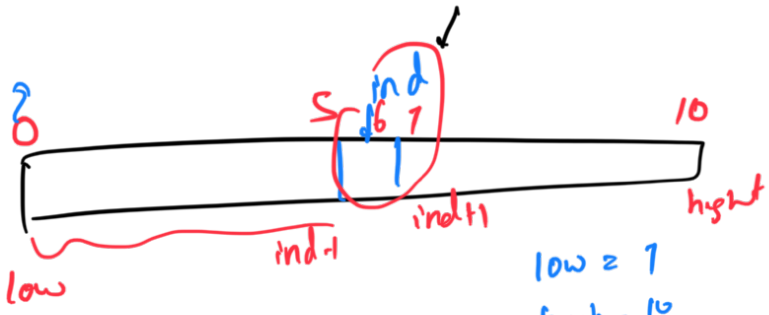
Inbuilt Libraries do use Quick sort.




```
void quickSort ( A[], low, high)
// Base Case  if( low ≥ high) return;
ind = partition ( A, low, high);
quickSort( A, low, ind-1);
quickSort( A, ind+1, high);
}
```



## Choices for Pivot

1) Last element : Arr[high]
2) First element : Arr[low]
3) Random element
4) Median of 1st, middle and last element

### Best Case

→ When the pivot is median of the array

$T(n) = 2T(\frac{n}{2}) + O(n)$

T.C: $O(n \log n)$

S.C: $O(\log n)$

### Worst Case

→ When pivot is either smallest or largest

$T(n) = T(n-1) + O(n)$

$T(n-1) = T(n-2) + O(n-1)$

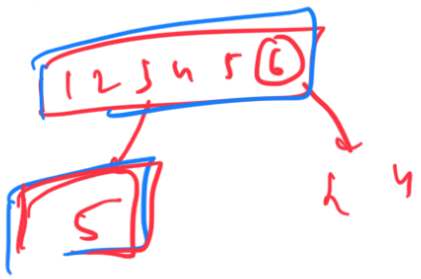
$T(n) = T(n-2) + n + n-1$

$T(n) = T(n-3) + n + (n-1) + (n-2)$

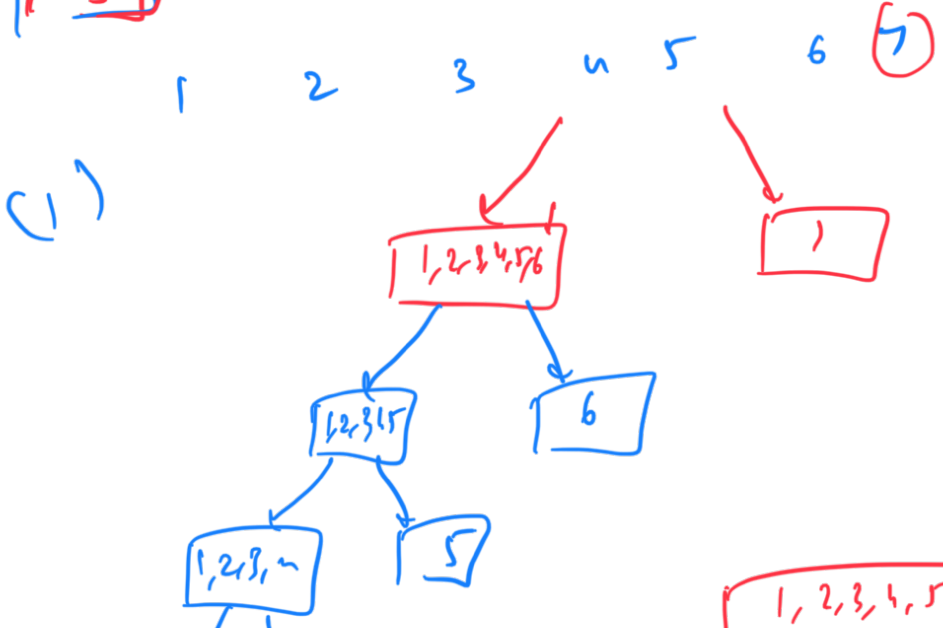
$T(n) = T(1) + n + n-1 + n-2 + n-3 \dots + 1$

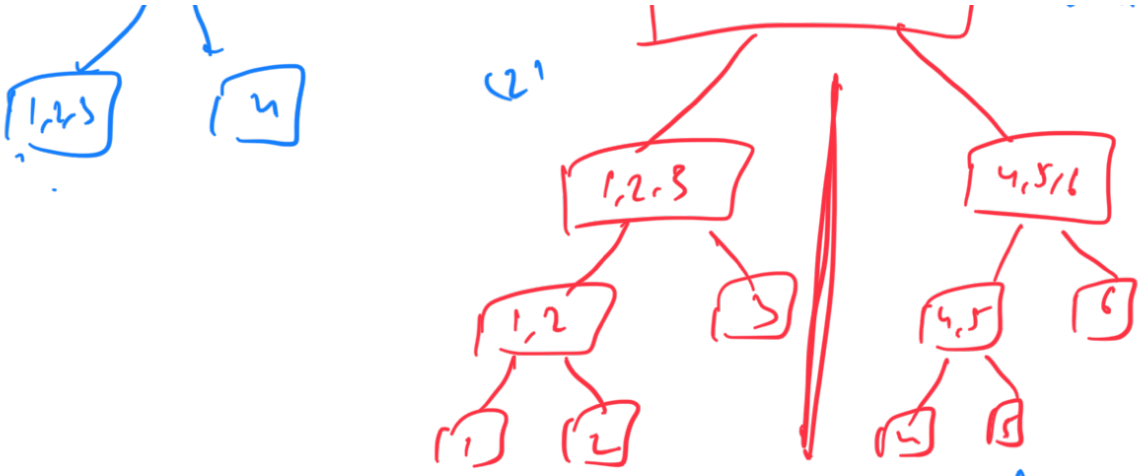
$T(n) = O(n^2)$

S.C: $O(n)$





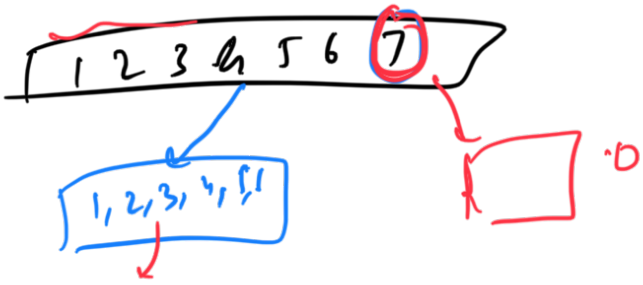




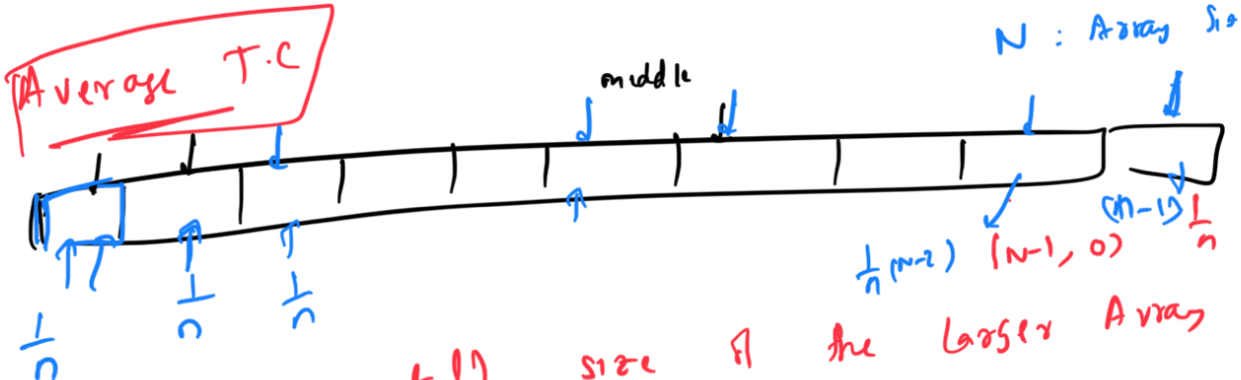

Height of Recursion : $O(\log n)$ Height of the free : $\log n$

$T(n)$ = Time for sorting an array of size $n$

$T(n) = n + T(n-1)$




S.C :




-) Average (Expected) size of the larger Array

$\frac{2}{n}[(n-1) + \frac{2}{n}(n-2) + \frac{2}{n}(n-3) + \cdots \frac{2}{n}(\frac{n}{2}) \; n(2)$

Expected = $\sum$ Probability $\times$ value

$$\frac{2}{n}\left[(n-1) + (n-2) + (n-3) + \cdots \frac{n}{2}\right]$$

$\simeq \frac{3N}{4} - \frac{1}{2}$.

Average size of the largest array.

$$T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right) + O(n)$$

$n \to \frac{3n}{4} \to \left(\frac{3}{4}\right)^2 n \to \left(\frac{3}{4}\right)^3 n \cdots \left(\frac{3}{4}\right)^k n$

$\downarrow$ 1

$\left(\frac{3}{4}\right)^k n = 1$

$n = \left(\frac{4}{3}\right)^k$

$\log n = k \log \frac{4}{3}$

$k = \log_{4/3} n$

$k = (\log n)$

No. of steps (Height of the recursion tree)

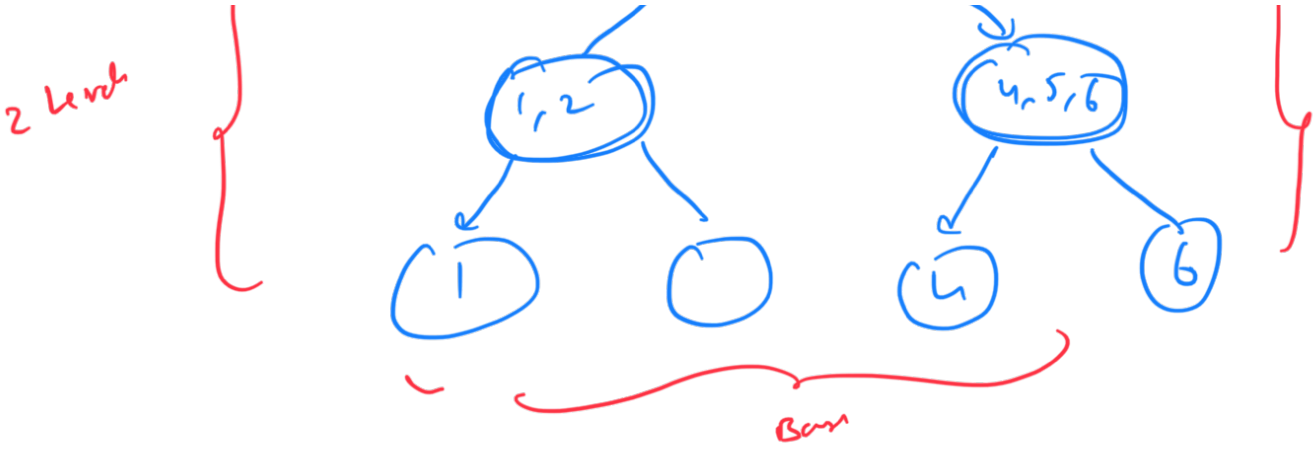
T.C : $\log n \times N$

T.C : $O(N \log n)$

Average T.C

Array is Sorted

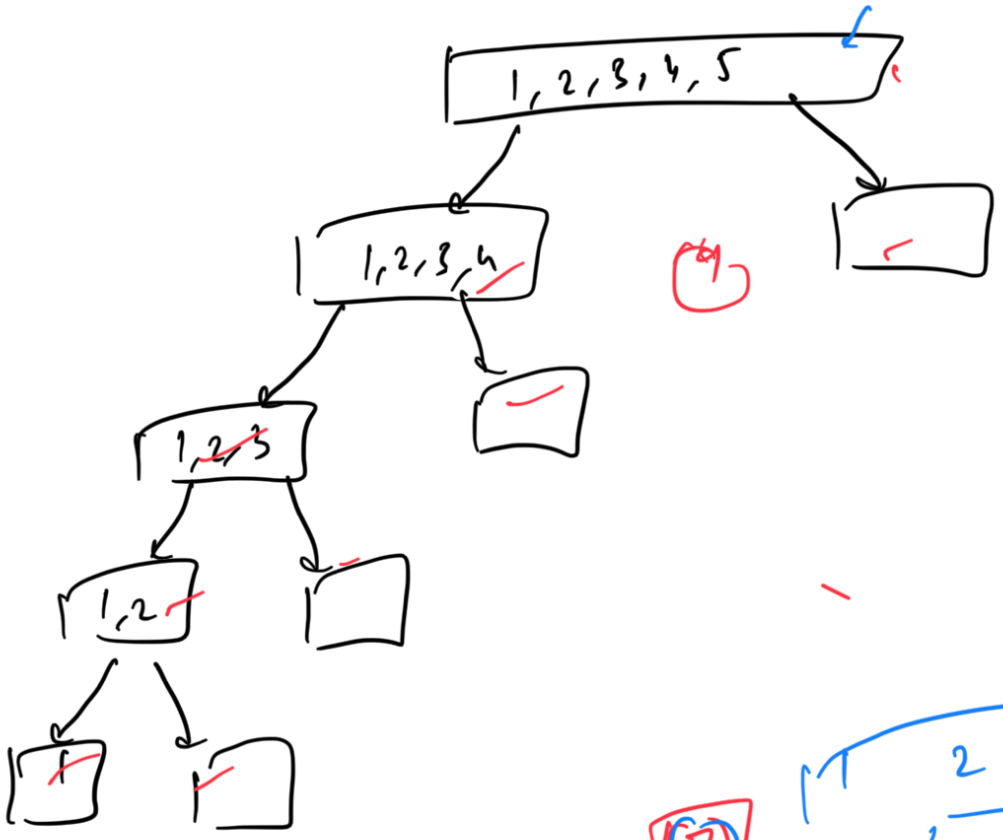
A = 1 2 3 4 5 6

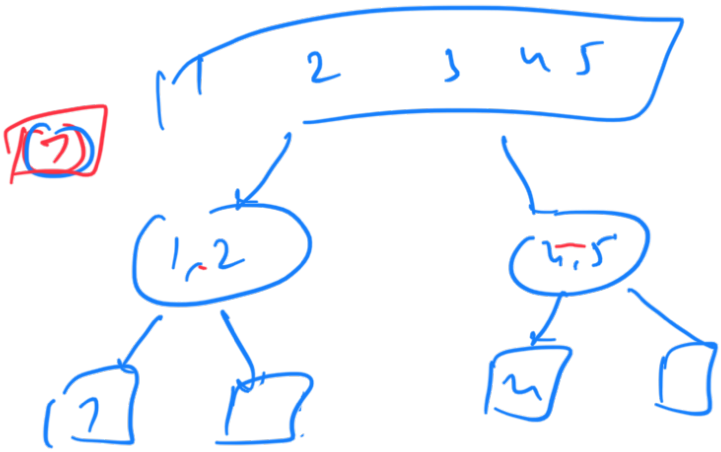



$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Master's theorem

$O(n \log n)$




Merge sort?

Quick Sort Better

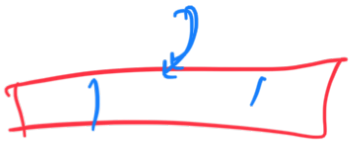
1) S.C of Merge Sort : O(n) (Allocating memory taking time)

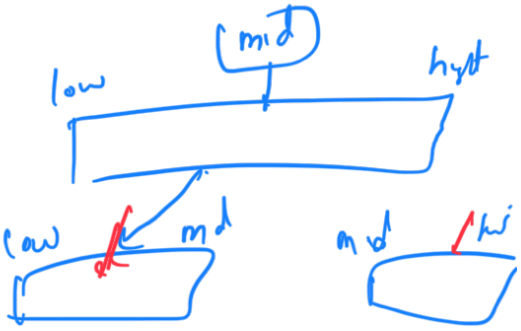
S.C of Quick Sort : O(logn)

2) Locality of Reference

Merge : Random Access

Quick Sort : Sequential Access (Better caching)




Heap Sort: O(nlogn)

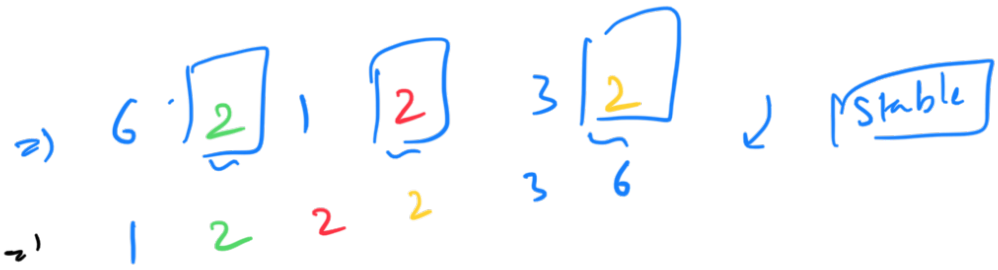
Quick Sort is seen to perform better than Merge Sort and Heap Sort.

O(nlogn) Average

Random Pivot

Tim-Sort

Quick Sort (Merge) + Insertion


## Bucket Sort :

Variation of Count sort

[10, 30, 567, 135]

→ Array elements are in the range [1-1000]

freq :

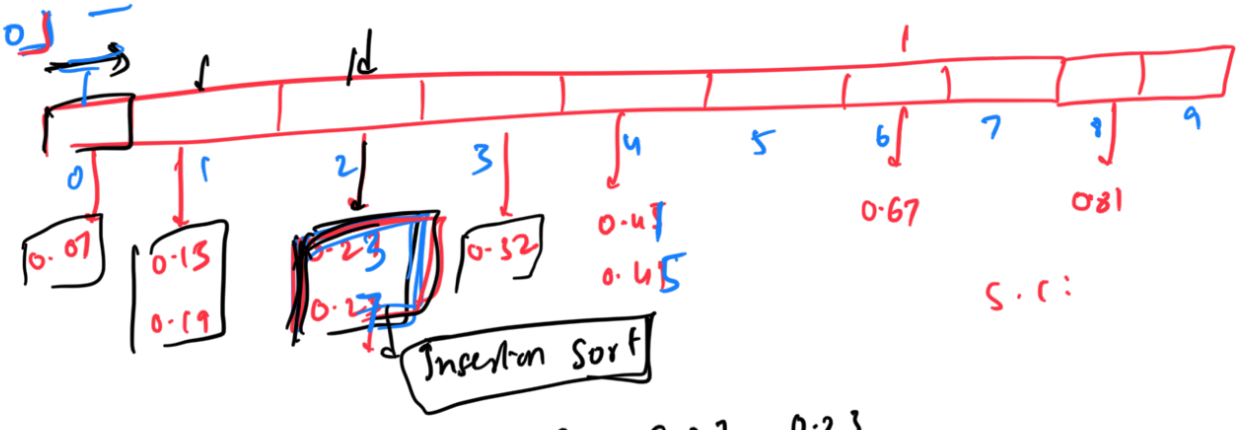

→ If the array elements are in the [0, 1]

A = [0.45, 0.27, 0.23, 0.67, 0.81, 0.15, 0.32, 0.19, 0.07, 0.41]

→ Map these floats to integer.

A[10]

→ [A[i] * 10]




S.C :

A = 0.07 0.15 0.19 0.27 0.23

→ N buckets ⇒ O(n) space

N ...

1 bucket ⇒ 1 element

A = [ ]

Range : [0, 1]

```
void bucketSort(float A[], int N) {
    vector<int> buckets[n];
    for(i=0; i<N; i++) {
        bucket_num = A[i] * N;
        buckets[bucket_num].push(A[i]);
    }
    for(i=0; i<N; i++)
        sort(buckets[i]);
    for(i=0; i<N; i++)
        // Print elements of buckets[i];
}
```

[0, 1]

A = (0.23 0.31 0.17 0.26)


(A[i] * 4)

(0, 3)

Range : [0 - 1]

Range of 4 * A[i] = [0 - 4]

0.91 * 4

3-6

0.2

0.2 * 4 = 0-8

100 elem => 100 buckets

0.24

Best Case T.C : $O(n)$

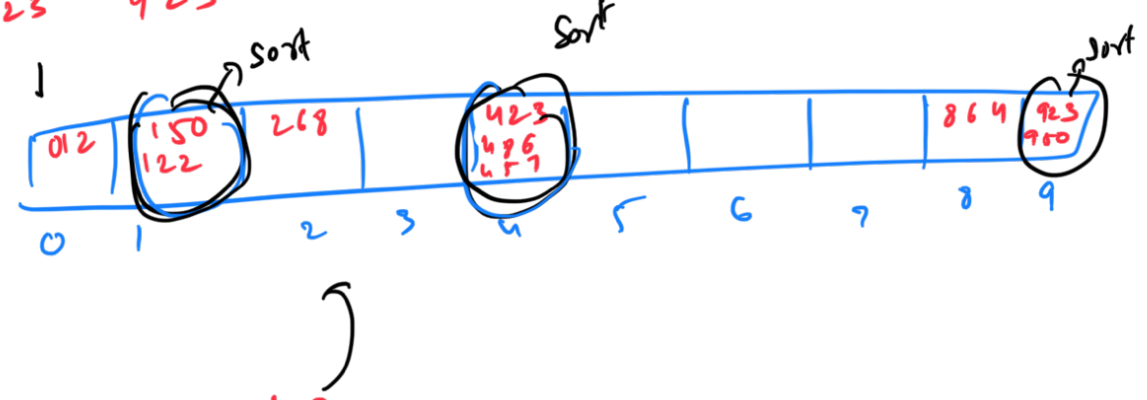
N elements ⇒ 1 bucket Each

Worst Case : $O(n^2)$ → Insertion

Merge Sort → $O(n \log n)$

Example

A = 423 923 150 268 012 950 122 864 486 457




Radix Sort:

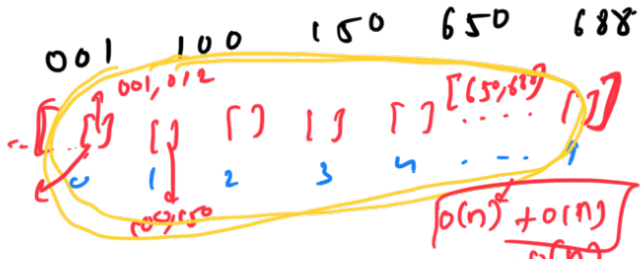
Variation of Bucket sort.

⇒ Gets rid of Sorting the buckets.

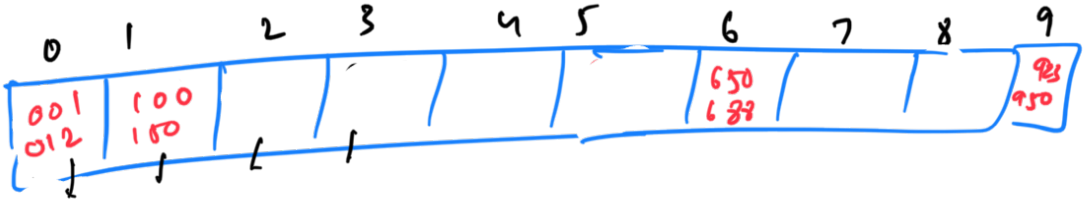
Any dgt: 0-9

[ ] [ ] [ ] . . . [ ]

A = 923 950 012 001 100 150 650 688




Buckets:




step1 A: 950 100 150 650 001 012 923 688

950 150 650 688

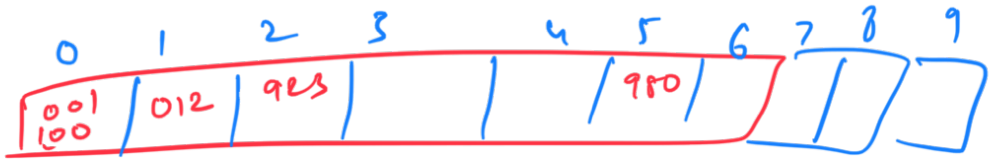
Step 2

A1 100 001 012 ...

Step3 : 001 012 100 150 650 681 923 950

O(n) + O(n) = O(n)

A2 923 950 012 001 100




Step1 : 012 001 100 923 950

Step2 : 001 100 012

A2 1015, 0002, 0034, 0656

=> No of digits of MAX Element.

T.C : O(n x d)

$\log_{10}^{(max)}$

No. of digits of Max element (n)

$\log_{10}^{n}$

(3 4 5 6 7 ~~8~~)

x = x, $\frac{x}{10}$, $\frac{x}{10^2}$ ..... $\left(\frac{x}{10^k}\right) > 1$

$\frac{x}{10^k} = 1$

$x = 10^k$

$k = \log_{10}^{x}$

T.C : O(N log N)

S.C : O(N)

long long int => 18

```
void radixSort ( A[], N ){
    int maxm = max(A[]);
    for(digit = 0; maxm>0; digit++){
        function( arr, n, digit);
        maxm = maxm/10;
    }
}

void function ( arr, n, digit) {
    vector<int> buckets[10];
    for(i=0; i<N; i++){
        b_num = (arr[i]/10^digit) % 10;
        buckets[b_num].push( arr[i]);
    }
    for(i=0; i<N; i++){
        // push Elements of bucket[i];
    }
}
```

digit = 0

digit = 0

$(453)/10^0$

453 561 479

$(453/10^1)\%10$

(45)

3 7 9 6 5

$(37965/10^3)\%10$

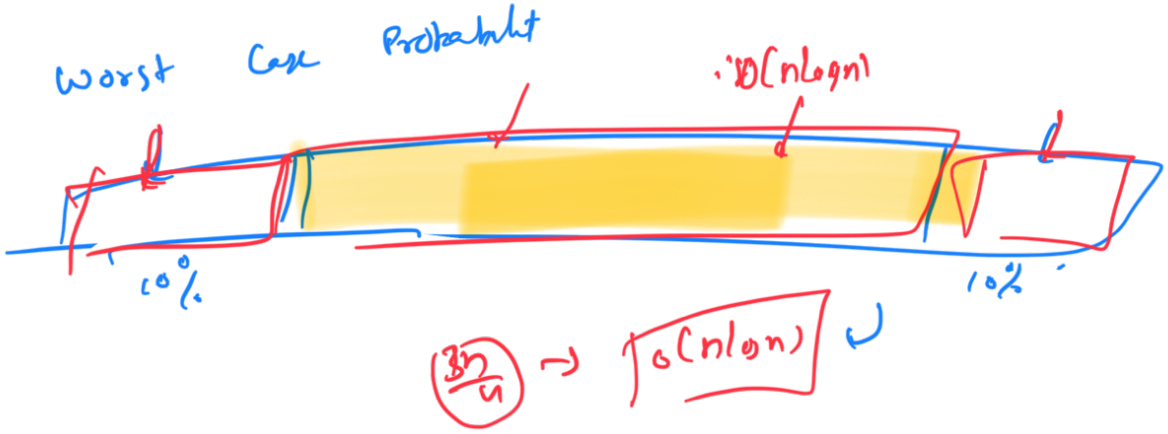
$(37)\%10 = 7$

$(N/10^{digit})\%10$

0-9

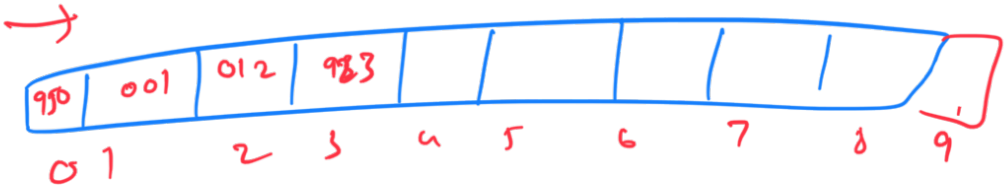
Hashmap

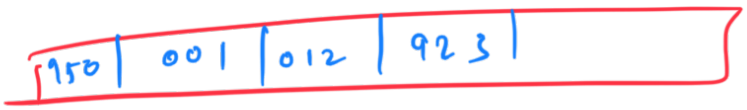
$\frac{2}{n}$




$\frac{3n}{4} \rightarrow O(n \log n)$

A = 923 950 012 001



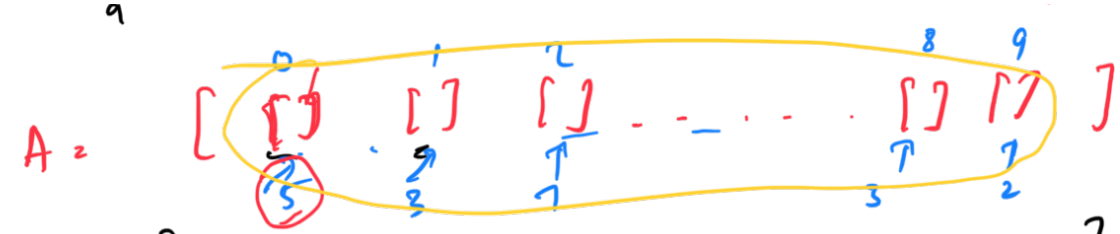


10 digit

0th
1st
2
3
.
.
.
n

1) Space : Allocate & Deallocate an array of size N: Mem

9




A = [ 100 , 350 . . . . . ]2

References

O(10) = O(1)

Array = [563 121 369 . . . ]

20

N = 20

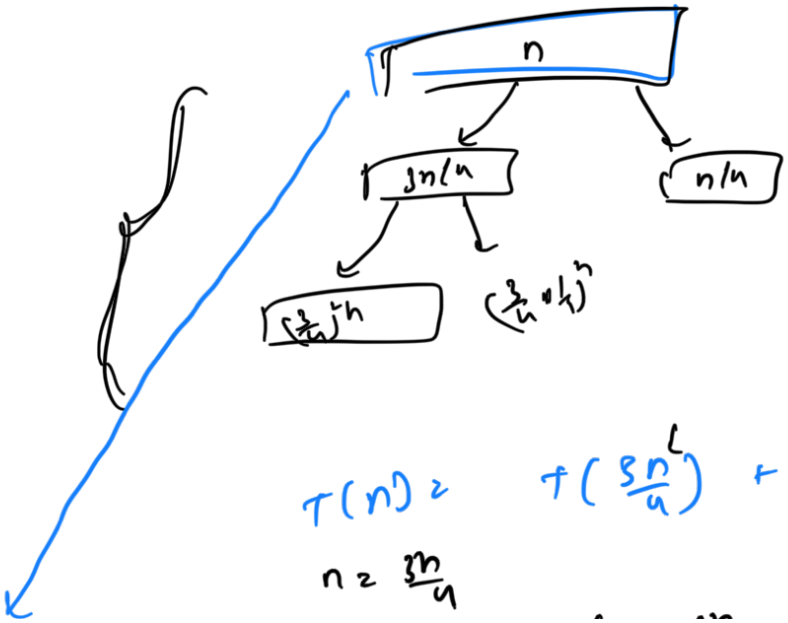
$\log_{10} n$

O(N) + O(10) + O(N)

$\left(\frac{3n}{4}\right)$

Expected val = Probability * value.

$T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{4}\right) + O(n)$




$n \to \frac{3}{4}n \to \left(\frac{3}{4}\right)^2 n \ldots\ldots 1$

$T(n) = T\left(\frac{3n}{4}\right)^L + T\left(\frac{n}{4}\right) + O(n)$

$n = \frac{3n}{4}$

```
quickSort(A, 0, n-1);
void quickSort(A[], int low, int high){
      if(low >= high)
            return;

      ind = partition(A, low, high);
      quickSort(A, low, ind-1);
      quickSort(A, ind+1, high);
return;
}
```

```
int partition(int A[], int low, int high){
      pivot = A[high];
      // Count no. of smaller elements
      for(int i = low; i <= high; i++)
            if(A[i] < pivot)
                  ind++;

      // Place the element in its right place
      swap(A[high], A[low + ind]);
      p1 = low , p2 = high;
      while(p1 < low + ind && p2 > low + ind){
            while(A[p1] < pivot)
                  p1++;
            while(A[p2] >= pivot)
                  p2--;
            swap(&A[p1], &A[p2]);
      }
return ind;
}
```