Quick Sort

The basic version of quick sort algorithm was invented by C. A. R. Hoare in 1960 and formally introduced quick sort in 1962. It is used on the principle of divide-and-conquer. Quick sort is an algorithm of choice in many situations because it is not difficult to implement, it is a good "general purpose" sort and it consumes relatively fewer resources during execution.

Advantages

- It is in-place since it uses only a small auxiliary stack.
- It requires only $n \log(n)$ time to sort n items.
- It has an extremely short inner loop
- This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

Disadvantages

- It is recursive. Especially if recursion is not available, the implementation is extremely complicated.
- It requires quadratic (*i.e.,* n^2) time in the worst-case.
- It is fragile i.e., a simple mistake in the implementation can go unnoticed and cause it to perform badly.

Algorithm

It works in two portion

1. QUICKSORT(A, p, r) – It do the division
2. PARTITION (A, p, r) – It is the subroutine of quicksort(A, p, r). It finds out the elements exact position

```
QuickSort(A, p, r)
{
        if(p==r)
          return(a[p])
        else
        {
          m=partition(A, p, r)
          Quicksort(A, p, m-1)
          Quicksort(A,m+1,r)
          return(A)
        }
}


Partition(A, p, q)

   x = A[p];
   i = p;
   for (j=i+1; j<=q; j++)
   {
         if( A[j] <= x)
          {
             i = i + 1;
             swap( a[i] , a[j] );
          }
   }
  swap ( a[i], a[p]);
  return(i);
}
```

Example of Quick Sort

p

40	80	96	25	54	68	16	10	35
1	2	3	4	5	6	7	8	9

i j

Scan list one by one element < pivot

Here x = A[p] = A[1] = 40

i = p = 1

j = i + 1 = 2

j=2 compare 80 and 40 80<=40 false no change

increment j, i.e. j=3

j=3 compare 96 and 40 96<=40 false

increment j, i.e. j=4

j=4 compare 25 and 40 25<=40 True

increment i, i.e. i=2 and exchange with j

40	25	96	80	54	68	16	10	35
1	2	3	4	5	6	7	8	9

increment j, i.e. j=5

j=5 compare 54 and 40 false

increment j, i.e. j=6

j=6 compare 68 and 40 false
increment j, i.e. j=7

j=7 compare 16 and 40 16<=40 True
increment i, i.e. i=3 and exchange with j

40	25	16	80	54	68	96	10	35
1	2	3	4	5	6	7	8	9

increment j, i.e. j=8
j=8 compare 10 and 40 10 <= 40 True
increment i, i.e. i=4 and exchange with j

40	25	16	10	54	68	96	80	35
1	2	3	4	5	6	7	8	9

increment j. i.e. j=9

j=9 compare 35 and 40 35 <= 40 True
increment i, i.e. i=5 and exchange with j

40	25	16	10	35	68	96	80	54
1	2	3	4	5	6	7	8	9

Now exchange A[i], A[p]

35	25	16	10	40	68	96	80	54
1	2	3	4	5	6	7	8	9

Now quick sort

35	25	16	10

Quick sort

68	96	80	54

On complete sorting

10	16	25	35	40	54	68	80	96

T(n) be the time complexity of quick sort algorithm on n-elements then Recurrence Relation:

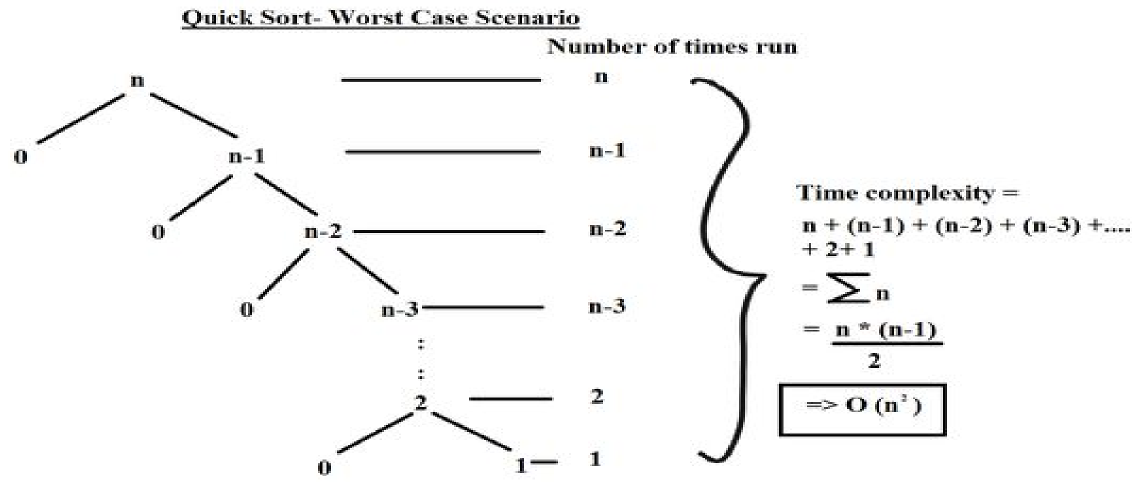
$$\mathbf{T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(P-m) + T(q-m) + n & \end{cases}}$$

Note: The running time of this algorithm depends on the partitioning of the large list into two sub-list and this partitioning is performed with the help of pivot element. This pivot element may partition the list in any one of the following case:

Worst-case partitioning: The partitioning occurs when pivot divides the n items list to n-1 and 0 partition resulting in unbalanced tree. The partitioning cost Θ(n) time. Since, the recursive call on array of size 0 returns T(0) = Θ(1). The recursive equation is :

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

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



Example- 10 20 30 40 50 60 70
It is the example which always give left gap

10 20 30 40 50 60 70
no smaller element so no i is incremented
7-element – 6 comparison – 1 swap

10 (20 30 40 50 60 70)
6 element – 5 comparison – 1 swap

20 (30 40 50 60 70)
5 element – 4 comparison – 1 swap
30 (40 50 60 70)
4 element – 3 comparison – 1 swap
40 (50 60 70)
3 element – 2 comparison – 1 swap
50 (60 70)
2 element – 1 comparison – 1 swap
60 (70)
1 element – 0 comparison

Total Comparison = n-1 + n-2 + n-3 + n-4+ ……..+1
=(n-1)(n-1+1)/2
=O(n^2)
(n-1) 6 passes for above problem

Total swap = $\underbrace{1+1+1+1+\ldots\ldots+1}_{\text{n-1 swap}}$

= O(n)

Stack size = n

Total space = input + stack size

= n + n

=O(n)

Time complexity of quick sort depends on comparison because comparisons are more

Best Case: Best case partitioning occurs when the pivot element divides the list into two sub-lists of equal size (Balanced tree).
The Recurrence Relation:

$$T(n) = 2T(n/2) + n$$

Stack size= logn

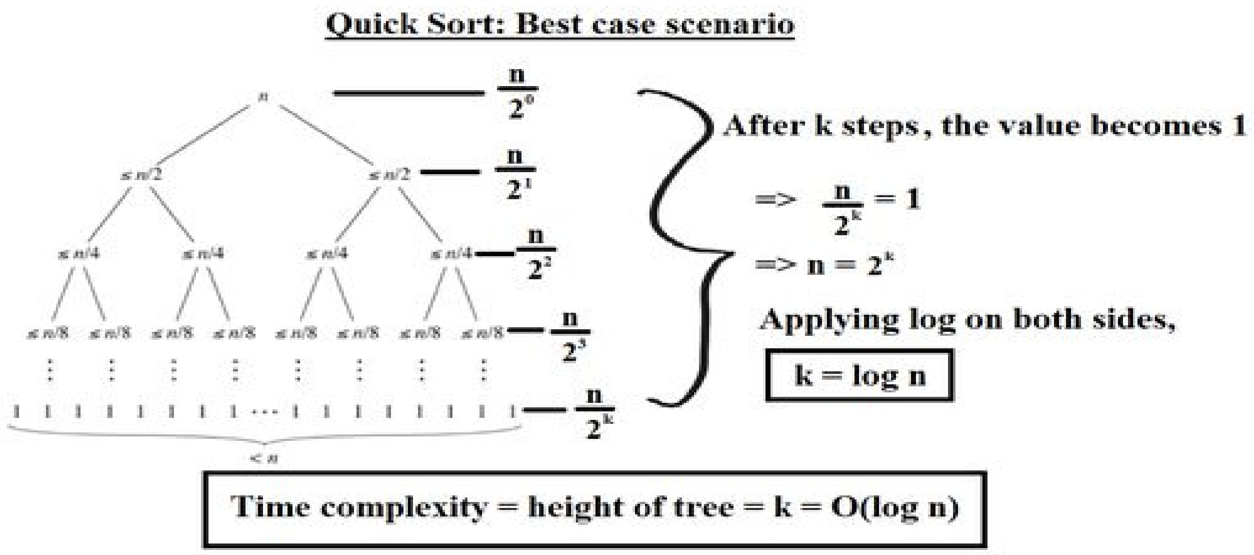
Total space = input + stack size

= n + logn

=O(n)

Average Case: This occurs when the sub-list size lies in between the best case and the worst case size of sub-list.

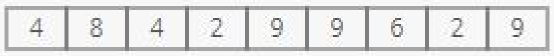
O(nlogn)


Quick Sort: Best case scenario
n
≤ n/2
≤ n/4
≤ n/8
1 1 1 1 1 1 1 1 ··· 1 1 1 1 1 1 1 1
< n
$\frac{n}{2^0}$
$\frac{n}{2^1}$
$\frac{n}{2^2}$
$\frac{n}{2^3}$
$\frac{n}{2^k}$
After k steps, the value becomes 1
$\Rightarrow \frac{n}{2^k} = 1$
$\Rightarrow n = 2^k$
Applying log on both sides,
$\boxed{k = \log n}$
$\boxed{\text{Time complexity = height of tree} = k = O(\log n)}$

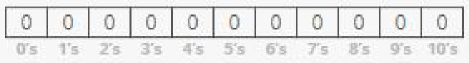
Sorting in Linear Time

Counting How Many Times Each Item Occurs

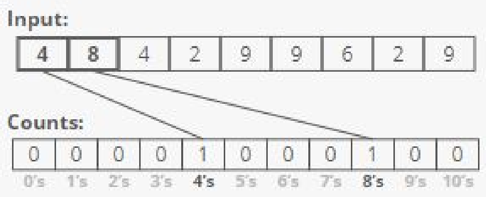
Say we have this array:



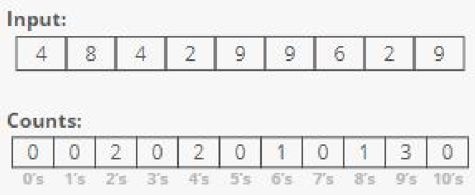
And say we know all the numbers in our array will be whole numbers between 0 and 10 (inclusive).
The idea is to count how many 0's we see, how many 1's we see, and so on. Since there are 11 possible values, we'll use an array with 11 counters, all initialized to 0.



We'll iterate through the input once. The first item is a 4, so we'll add one to `counts[4]`. The next item is an 8, so we'll add one to `counts[8]`.

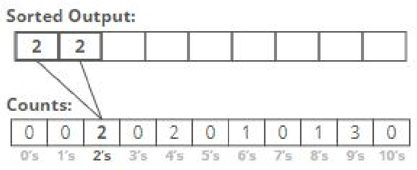


And so on. When we reach the end, we'll have the total counts for each number:

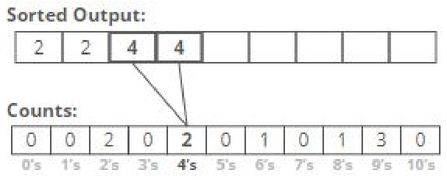


Building the Sorted Output

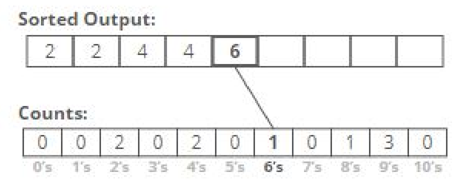
Now that we know *how many* times each item appears, we can fill in our sorted array. Looking at `counts`, we don't have any 0's or 1's, but we've got two 2's. So, those go at the start of our sorted array.



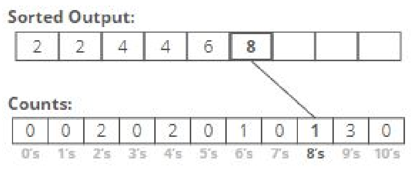
No 3's, but there are two 4's that come next.



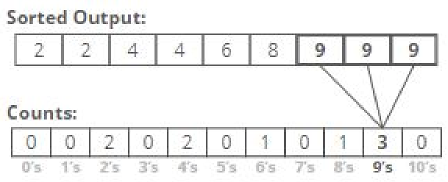
After that, we have one 6,



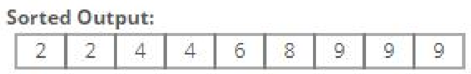
one 8,



and three 9's



And, with that, we're done!



Algorithm

```
countingSort(array, size)

  max <- find largest element in array

  initialize count array with all zeros

  for j <- 0 to size
```

```
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Complexity

Counting sort takes **$O(n+k)$ time and $O(n+k)$ space**, where n is the number of items we're sorting and k is the number of possible values.

We iterate through the input items twice—once to populate `counts` and once to fill in the output array. Both iterations are $O(n)$ time. Additionally, we iterate through `counts` once to fill in `nextIndex`, which is $O(k)$ time.

The algorithm allocates three additional arrays: one for `counts`, one for `nextIndex`, and one for the output. The first two are $O(k)$ space and the final one is $O(n)$ space.

In many cases cases, k is $O(n)$ (i.e.: the number of items to be sorted is not asymptotically different than the number of values those items can take on. **Because of this, counting sort is often said to be $O(n)$ time and space**.

Strengths:

- **Linear time**. Counting sort runs in $O(n)$ time, making it asymptotically faster than comparison-based sorting algorithms like quicksort or merge sort.

Weaknesses:

- **Restricted inputs**. Counting sort only works when the range of potential items in the input is known ahead of time.
- **Space cost**. If the range of potential values is big, then counting sort requires a lot of space (perhaps more than O(n)$O(n)$).

Complexity	
Worst case time	O(n)
Best case time	O(n)
Average case time	O(n)
Space	O(n)