Quick sort

Quick sort/Partition Exchange sort

- ***It uses the concept of divide and Conquer***
- There are two main parts of the algorithm:

1. ***Choosing a pivot element and partitioning the array***

- Either the first element or any random element of array is chosen as a pivot element
- Array is partitioned such that pivot element is placed at its correct position.
- Elements less than equal to pivot are placed to the left of pivot element and the elements greater than pivot are placed to the right.

2. ***Applying quicksort again on partitioned arrays***

- After partitioning we get two arrays
- Each array is again partitioned in the same way.
- Recursively apply partitioning till the array has only one element.

Cont…

- Suppose we have an array X of size n
- We choose an element ***a*** from a specific position called as pivot element
- Elements of X are partitioned such that ***a*** is placed into position j and the following conditions hold good:

1. Each of the elements in positions through 0 to j-1 are less than equal to ***a***
2. All the elements in positions through j+1 to n-1 are greater than ***a***

Note: It is obvious that if the above two conditions hold good for the pivot element a, then jth position is the right position for this element

Partitioning algorithm

- Given an array x of size n and assuming first element as pivot element (Input)
- A new arrangement of array x of size n, in which pivot element ***a*** is placed at the right position, and all the elements less than equal to pivot are placed to the **left of pivot** and elements greater than pivot are placed to the **right of pivot**. (Output)
- **Step 1** − Choose the ***lowest index value*** element as pivot (lb- lowest index of array X, ub-highest index of array X)
- **Step 2** − Take two variables ***down*** and ***up*** to point to **lowest index** and **highest index** element of the array respectively
- **Step 3** − Repeatedly increase ***down*** pointer by one position until x[down]>a (keep moving till we have x[down]<=a)
- **Step 4 -** Repeatedly decrease ***up*** pointer by one position until x[up]<=a (keep moving till we have x[up]> a)
- **Step 5** − If up >down interchange x[down] and x[up]
- **Step 6** − Steps 3 to 5 are repeated until the condition in step 5 fails (means up<=down) and at this point x[up] is interchanged with x[lb] which is equal to ***a*** and **j** is set to **up**.

Example

24 60 85 1 20 15 11 100 75 (array X)

- N = 9, lb = 0, ub =8, down=0, up =8, pivot element a = 24
- We will start incrementing down and stop at down =1 because 60 >24
- We will start decrementing up and stop at up =6 because 11<=24
- Interchange X[1] by X[6]– 24 11 85 1 20 15 60 100 75
- incrementing down and stop at down =2 because 85 >24
- decrementing up and stop at up = 5 because 15<=24
- Interchange X[2] by X[5]– 24 11 15 1 20 85 60 100 75

Example

24 60 85 1 20 15 11 100 75 (array X) Original array

24 11 15 1 20 85 60 100 75 (now after making two interchanges) (down =2; up= 5)

- incrementing down and stop at down =5 because 85 >24
- decrementing up and stop at up = 4 because 20<=24
- Since up<=down, interchange x[up] with x[lb]:

    20 11 15 1 $\boxed{24}$ **85 60 100 75**

- Above array is partitioned in two subarrays X[0] to X[3] and X[5] to X[8]
- Now we will apply partition algorithm on these two arrays taking ***X[0] pivot element for first array*** and taking ***X[5] pivot element for second array***

Applying partition on subarrays

- Array1: ***20 11 15 1*** (lb =0 ub = 3, a= 20, down =0 up = 3)
- Array2: ***85 60 100 75*** (lb =5 ub = 8, a =85, down =5 up = 8)
- Array1: 20 11 15 1 (lb =0 ub = 3, a= 20, down =0 up = 3)
- Incrementing down… … = 3
- Decrementing up.. , will not be decremented. So up = 3
- Hence x[up] will be exchanged with x[lb] and we will have:
- [1 11 15] [20]
- Array2: 85 60 100 75 (lb =5 ub = 8, a =85, down =5 up = 8)
- Incrementing down… … = 7 and up will not be decremented. So up = 8
- Interchanging x[down] and x[up]: 85 60 75 100, next iteration down =8, up=7, interchanging x[up] and x[lb]: [75 60] 85 [100]

Partition function

```
Void partition(int x[],int lb,int ub, int *pj)
{ int a, down, up, temp;
 a = x[lb]; up =ub;down = lb;
While(down<up) {while(x[down] <=a && down <ub) down++;
                while(x[up] >a) up--;
                if(down <up)   swap(x[down],x[up])
               }
            x[lb] = x[up]; x[up] =a; *pj = up;
}
```

Quick sort algorithm

```
 if(lb>=ub) return;
Partition(x,lb,ub,j);
Quick(x, lb, j-1);
Quick(x,j+1,ub);
```

Quicksort Sorting efficiency..

- We know that sorting efficiency depends upon the arrangement of input data and input data may be already sorted, nearly sorted, completely random, in reverse order or nearly reverse.
- Examine Quick sort:
- **How many passes of partition will be there if:**

1. The arrangement of data is such that **pivot element** partitions the array into two almost equal arrays
2. Pivot element does not partition equally or almost equally

Quick sort complexity

- ***Best case*** (Pivot divides the array into two equal subarrays)
- $O(n\log_2 n)$

$n + 2*n/2 \quad +4 *n/4 \quad +\dots\dots\dots\; n*(n/n)$

$= n + n + \dots\; n$ (m terms) $= n\log_2 n$

- Worst case (Data is sorted or in reverse order)
- $O(n^2)$

$=n + (n-1) + (n-2) +\dots.\; 1 = n(n-1)/2$

- Average case $O(n\log_e n)$ (Random data)
- Requires more space due to recursion and depends upon the number of nested recursive calls and the size of stack.

Few ways to choose pivot elements…

- Several choices have been found to improve the efficiency of quicksort by creating nearly balanced subfiles
- First Technique: Take the median of first, last and middle element i.e median of x[lb], x[ub] and x[(lb+ub)/2] this median of three value is generally closer to the median of the file/subfile being partitioned
- Second Technique: Take first pivot as median and then subsequently use mean as pivot of each subfile.
- Other techniques: a. Use the middle element of file as pivot element

  b. Choose a random element