Reg.No : 24BCE0554
Name : Partha Pratim Gogoi
Topic : Quick Sort

Algorithm:

```
1   partition(arr, low, high)                                  [T.C: O(m)]
    1.1  pivot = arr[high]
    1.2  i = low-1
    1.3  for j in low→high                                   [T.C: O(m)]
       1.3.1   if (arr[j] <= pivot) then
       1.3.2   swap(arr[++i], arr[j])                          [T.C: O(1)]
    1.4  swap(arr[i+1], arr[high])
    1.5  return i+1
2   randomPivot(arr, low, high)                                [T.C: O(m)]
    2.1  pivotIndex = low + rand()%(high-low+1)              [T.C: O(1)]
    2.2  swap(arr[pivotIndex], arr[high])                    [T.C: O(1)]
    2.3  return partition(arr, low, high)                    [T.C: O(m)]
3   propagateQS(arr, low, high)               [T.C: T(m) = T(l)+T(r)+3O(m)]
    3.1  if (low < high) then
       3.1.1   p = randomPivot(arr, low, high)                 [T.C: O(m)]
       3.1.2   propagateQS(arr, low, p-1)         [T.C: T(m) = T(l)+O(m)]
       3.1.3   propagateQS(arr, p+1, high)        [T.C: T(m) = T(r)+O(m)]
4   quickSort(arr)                            [T.C: T(m) = T(l)+T(r)+3O(m)]
    4.1  if (arr.size() <= 1) then return
    4.2  propagateQS(arr,0,arr.size-1)[T.C: T(m) = T(l)+T(r)+3O(m)]
5   generateRandomSeed()                                       [T.C: O(1)]
6   quickSort(arr)

    where,
      m = high-low + 1              [index values of each sub-array]
      l = left sub-array size       [p-1 - low]
      r = right sub-array size      [high - p+1]
```

Time Complexity:

NOTE:

- T(0) = O(1)
  because ‘no time taken’ is same as ‘constant time taken’
- partition() always takes O(n) time overall

Base recurrence relation

T(m) = T(l) + T(r) + 3O(m)   [NOTE: l+r = m-1]

* Worst Case: Left/Right sub-array is empty

- l = 0
- r = m-1

T(m) = O(1) + T(m-1) + O(m)   [3O(m) converts to O(m)]

T(m) = T(m-1) + O(m)

Sum(T(m)) - Sum(T(m-1)) = Sum(O(m))

T(m) - T(0) = Sum(O(m))

T(m) - O(1) = O(n)   [m+…+2+1 = n]

Sum(T(m)) - Sum(O(1)) = Sum(O(n))

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

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

* Average Case: Left&Right sub-array are not empty and not equal

- l ~ m/2   (balanced on average)
- r ~ m/2   (balanced on average)

T(m) = 2T(m/2) + O(m)   [3O(m) converts to O(m)]

T(m) = 2T(m/2) = O(n)   [Refer NOTE]

By Master’s Theorem,

T(n) = O(nlogn)

* Best Case: Left&RIght sub-array are not empty and equal

- l = m/2   (exactly balanced)
- r = m/2   (exactly balanced)

T(m) = 2T(m/2) + O(m)   [3O(m) converts to O(m)]

T(m) = 2T(m/2) + O(n)   [Refer NOTE]

By Master’s theorem,

T(n) = O(nlogn)

Total Time Complexity

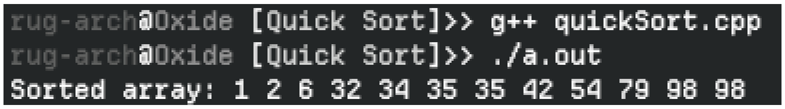
Avg/Best Case : O(nlogn)

Worst Case : $O(n^2)$

Source Code:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
using namespace std;

////////////////////////////////////
/// Parition Scheme - Main logic ///
////////////////////////////////////
int partition(vector<int>& arr, int low, int high) {
   // ---------------------------- //
   // Intialize starting variables //
   // ---------------------------- //
   int pivot = arr[high];                              // Random
element set as the last element
   int i = low-1;

   // ---------------------------------------- //
   // Elements smaller than pivot on left side //
   // ---------------------------------------- //
   for (int j=low; j<high; j++) {
       if (arr[j] <= pivot) {
           swap(arr[++i], arr[j]);                     // This is not
sorting the array
       }                                               // segregating
into higher & lower (than pivot)
   }
   // ------------------------------------------------ //
   // 0 ... i | i+1 (insert pivot here) | i+2 ... high //
   // ------------------------------------------------ //
   swap(arr[i+1], arr[high]);                          // Put the pivot
where it belongs by swapping it in
   return i+1;                                         // Correct pivot
index
}

/////////////////////////////////
/// Select random pivot index ///
/////////////////////////////////
int randomPivot(vector<int>& arr, int low, int high) {
   // -------------------------------------------- //
   // Select random pivot and swap to end of array //
   // -------------------------------------------- //
   int pivotIndex = low + rand() % (high-low+1);       // low + [0,
high-low+1] = [low, high]
   swap(arr[pivotIndex], arr[high]);
   // ---------------------- //
   // Apply partition scheme //
   // ---------------------- //
   return partition(arr,low,high);                     // Returns the
correct index for pivot value
}

///////////////////
/// Propagation ///
///////////////////
void propagateQS(vector<int>& arr, int low, int high) {
   if (low < high) {
       int p = randomPivot(arr, low, high);

       propagateQS(arr, low, p-1);                     // Left sub-
array
       propagateQS(arr, p+1, high);                    // Right sub-
array
   }
}

//////////////////
/// Initiation ///
//////////////////
void quickSort(vector<int>& arr) {
   if (arr.size() <= 1) return;

   propagateQS(arr, 0, arr.size()-1);
}

int main() {
   vector<int> array = {1,79,2,35,6,98,34,32,98,42,54,35};

   // Seed for random number
   srand(time(NULL));

   quickSort(array);
   cout << "Sorted array: ";
   for (int el : array) {
       cout << el << " ";
   }
   cout << endl;

   return 0;
}
```

Sample Output:



```
rug-arch@Oxide [Quick Sort]>> g++ quickSort.cpp
rug-arch@Oxide [Quick Sort]>> ./a.out
Sorted array: 1 2 6 32 34 35 35 42 54 79 98 98
```