# Can QuickSort be implemented in O(nLogn) worst case time complexity?

The worst case time complexity of a typical implementation of QuickSort is O($n^2$). The worst case occurs when the picked pivot is always an extreme (smallest or largest) element. This happens when input array is sorted or reverse sorted and either first or last element is picked as pivot.

Although randomized QuickSort works well even when the array is sorted, there is still possibility that the randomly picked element is always an extreme. Can the worst case be reduced to O(nLogn)?

The answer is yes, we can achieve O(nLogn) worst case. The idea is based on the fact that the median element of an unsorted array can be found in linear time. So we find the median first, then partition the array around the median element.

Following is C++ implementation based on above idea. Most of the functions in below progran are copied from K’th Smallest/Largest Element in Unsorted Array | Set 3 (Worst Case Linear Time)

```
/* A worst case O(nLogn) implementation of quicksort */
#include<cstring>
#include<iostream>
#include<algorithm>
#include<climits>
using namespace std;

// Following functions are taken from http://goo.gl/ih05BF
int partition(int arr[], int l, int r, int k);
int kthSmallest(int arr[], int l, int r, int k);

/* A O(nLogn) time complexity function for sorting arr[l..h] */
void quickSort(int arr[], int l, int h)
{
    if (l < h)
    {
        // Find size of current subarray
        int n = h-l+1;

        // Find median of arr[].
        int med = kthSmallest(arr, l, h, n/2);

        // Partition the array around median
        int p = partition(arr, l, h, med);

        // Recur for left and right of partition
        quickSort(arr, l, p - 1);
        quickSort(arr, p + 1, h);
    }
}
```

```
// A simple function to find median of arr[].  This is called
// only for an array of size 5 in this program.
int findMedian(int arr[], int n)
{
    sort(arr, arr+n);  // Sort the array
    return arr[n/2];   // Return middle element
}

// Returns k'th smallest element in arr[l..r] in worst case
// linear time. ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        int n = r-l+1; // Number of elements in arr[l..r]

        // Divide arr[] in groups of size 5, calculate median
        // of every group and store it in median[] array.
        int i, median[(n+4)/5]; // There will be floor((n+4)/5) groups;
        for (i=0; i<n/5; i++)
            median[i] = findMedian(arr+l+i*5, 5);
        if (i*5 < n) //For last group with less than 5 elements
        {
            median[i] = findMedian(arr+l+i*5, n%5);
            i++;
        }

        // Find median of all medians using recursive call.
        // If median[] has only one element, then no need
        // of recursive call
        int medOfMed = (i == 1)? median[i-1]:
                                 kthSmallest(median, 0, i-1, i/2);

        // Partition the array around a random element and
        // get position of pivot element in sorted array
        int pos = partition(arr, l, r, medOfMed);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// It searches for x in arr[l..r], and partitions the array
// around x.
int partition(int arr[], int l, int r, int x)
{
    // Search for x in arr[l..r] and move it to end
    int i;
    for (i=l; i<r; i++)
        if (arr[i] == x)
           break;
    swap(&arr[i], &arr[r]);

    // Standard partition algorithm
    i = l;
    for (int j = l; j <= r - 1; j++)
    {
        if (arr[j] <= x)
        {
            swap(&arr[i], &arr[j]);
            i++;
        }
    }
    swap(&arr[i], &arr[r]);
    return i;
}

/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1000, 10, 7, 8, 9, 30, 900, 1, 5, 6, 20};
    int n = sizeof(arr)/sizeof(arr[0]);
    quickSort(arr, 0, n-1);
    cout << "Sorted array is\n";
    printArray(arr, n);
    return 0;
}
```

Run on IDE

Output:

```
Sorted array is
1 5 6 7 8 9 10 20 30 900 1000
```

**How is QuickSort implemented in practice – is above approach used?**

Although worst case time complexity of the above approach is O(nLogn), it is never used in practical implementations. The hidden constants in this approach are high compared to normal Quicksort. Following are some techniques used in practical implementations of QuickSort.

1) Randomly picking up to make worst case less likely to occur (Randomized QuickSort)

2) Calling insertion sort for small sized arrays to reduce recursive calls.

3) QuickSort is tail recursive, so tail call optimizations is done.

So the approach discussed above is more of a theoretical approach with O(nLogn) worst case time complexity.

This article is compiled by **Shivam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Company Wise Coding Practice** **Topic Wise Coding Practice**

10 Comments Category: Sorting Tags: Sorting

## Related Posts:

- Sort an array according to count of set bits
- Rearrange positive and negative numbers
- Find Surpasser Count of each element in array
- BogoSort or Permutation Sort
- Odd-Even Sort / Brick Sort
- Sorting Vector of Pairs in C++ | Set 2 (Sort in descending order by first and second)
- Sorting Vector of Pairs in C++ | Set 1 (Sort by first and second)
- Gnome Sort

(Login to Rate and Mark)

4.5 Average Difficulty : **4.5/5.0**
Based on **6** vote(s)

☐ Add to TODO List
☐ Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

**10 Comments** **GeeksforGeeks** **Tejas Joshi**

**Recommend** **Share** **Sort by Newest**

Join the discussion…

**P!=NP** • a month ago
This is really not a good implementation of deterministic qsort....no way near to O(n)
:::::Results ::: //Input generated random !
Input : 10^5 : 12.89 sec
Input: 10^6 : SORRY I DON'T think this program will HALT any time soon :)
//I waited for 15min then ..."ctrl+c"
BUT if I run qsort using simple LOMUTO partition scheme ...
for input 10^6 qsort its taking like 19-20 sec. (w/t optimization)

and stdlib qsort 0.2333 sec (^0^)
Reply • Share ›

**Cherish** • 2 years ago
This algorithm rocks in theory, but sucks in practice. Linear time median finding has too much

overhead, even though it's O(N)

3 ^ | v • Reply • Share ›

**Rajdeep Podder** • 2 years ago

There are several ways like.

1. Shuffle the list using some linear time shuffling algorithm e.g. Knuth Shuffle and chose first element as pivot

2. Use method called "Tukey's ninther" as pivot choosing mechanism.
http://www.johndcook.com/blog/...

^ | v • Reply • Share ›

**Nishant** • 2 years ago

The complexity is controlled by partition() method of the quicksort.

In above code, below code snippet forms the basis of complexity:
(I am incrementing a glo.bal value 'e' to count the number of times of execution.

```
// Standard partition algorithm
i = l;
for (int j = l; j <= r - 1; j++)
{
e++;
if (arr[j] <= x)
{
swap(&arr[i], &arr[j]);
i++;
}
}
```

For the given i/p complexity(value of e)= O(60)
which is way over nlogn = 11log11 =~ 11*4=44 >60

So it fails the test

^ | v • Reply • Share ›

**Nishant** → Nishant • 2 years ago

Below solution(in java) works better than your code, even if random swap is removed(55 complexity):

```
public void qsort(int l, int u){

if(l>=u) return;

int i, j;
int t;

int randNum = l+rand.nextInt(u-l+1);
swap(l, randNum);
```

```
t = x[l];
i = l;
j = u+1;

while(true) {

do{ i++; }while(i <= u && x[i] < t);
```

see more

• Reply • Share ›

**Nishant** • 2 years ago

Best way is to:
1) use randomized quick sort
2) use 2 way comparison in each recursive call
3) for lesser length arr perform insertion sort, don't divide further.

2 • Reply • Share ›

**Balaji** Nishant • 2 years ago

Rightly pointed out, but when it comes to Randomized algorithms it doesn't fall under worst case analysis rather a probabilistic analysis usually expected time, the question is on worst case analysis.

• Reply • Share ›

**Guest** Balaji • 2 years ago

worst case happens, when array is already sorted....in that case normal quick sort complexity will be n-square.

Using 2 way comparison, the complexity will be even better of O(n) complexity.

• Reply • Share ›

**Santhosh Kumar Devaraj Karthik** • 2 years ago

Usually one of the ways to make it work in nlogn time is to shuffle the array before sorting. This would reduce the instance where the partition element is the least element in the array at that point and every iteration sorts just that element resulting in quadratic complexity. I would suggest watching Rob Sedgewick's coursera classes where he talks in detail about this drawback with quick sort.

• Reply • Share ›

**Shivam** Santhosh Kumar Devaraj Karthik • 2 years ago

What you are saying is effectively equivalent to randomly choosing the pivot

Refer https://www.youtube.com/watch?... at 45:55

• Reply • Share ›