CS 514

Assignment 2 – Implementing and Analyzing Divide and Conquer Sorting Algorithms

report2.txt

Q1. Implement the mergeSort and quickSort algorithms to sort a set of integers in increasing order. You must add your own test cases.

Answer-

Please find the implementation of merge sort and quicksort (with first element as pivot and with randomized pivot) which sort a set of integers in increasing order.

```
import random # for randomized pivot


#implementation of merge sort algorithm
def merge_sort(list1):
    n = len(list1)
    if n <= 1:
        return list1
    return merge_sorted_lists(merge_sort(list1[:n//2]),
merge_sort(list1[n//2:]))


def merge_sorted_lists(sorted1, sorted2):
    if not sorted1 or not sorted2:
        return sorted1 + sorted2
    merged_list = []
    ind1, ind2 = 0, 0
    len1, len2 = len(sorted1), len(sorted2)
    while ind1 < len1 or ind2 < len2:
        if ind1 == len1 or (ind2 != len2 and sorted1[ind1] > sorted2[ind2]):
            merged_list.append(sorted2[ind2])
            ind2 += 1
        else:
            merged_list.append(sorted1[ind1])
            ind1 += 1
    return merged_list
```

```
#Implementation of quick sort algorithm
#When the pivot is the first element in the list
def quick_sort1(list1,l,h):
    if l < h:
        p = partition(list1,l,h)
        quick_sort1(list1,l,p)
        quick_sort1(list1,p+1,h)

#When the pivot the chosen randomly
def quick_sort2(list1, left, right):
    if left < right:
        pivot_index = left + random.randint(0, right - left)
        pivot_index = partition(list1, left, right, pivot_index)
        quick_sort2(list1, left, pivot_index - 1)
        quick_sort2(list1, pivot_index + 1, right)

def partition(list1,l,h):
    pivot = list1[l]
    i = l - 1
    j = h + 1

    while True:
        i = i + 1
        while list1[i] < pivot:
            i = i + 1
        j = j - 1
        while list1[j] > pivot:
            j = j - 1
        if i >= j:
            return j
        list1[i],list1[j] = list1[j],list1[i]

if __name__ == '__main__':
    #testcases
    arr0 = [197]
    arr1 = [49, 146, 88, 199, 35, 37, 189, 113, 111, 34, 24, 182, 142, 75,
155]
    arr2 = [17, 155, 58, 185, 94, 129, 167, 73, 76, 162, 183, 188, 87, 61,
134, 160, 6, 169, 79, 89, 110, 116, 36, 123,
            99, 180, 136, 50, 19, 37, 130, 25, 144, 55, 85, 176, 121, 146,
12, 88, 1, 39, 26, 112, 9, 86, 54, 143, 63,
            30, 93, 137, 101, 21, 97, 126, 107, 179, 114, 72, 64, 145, 150,
174, 5]
    arr3 = [114, 157, 145, 74, 27, 98, 194, 58, 93, 70, 156, 88, 56, 161, 69,
51, 80, 186, 125, 17, 126, 18, 191, 104,
            32, 111, 190, 22, 160, 77, 81, 76, 195, 85, 124, 14, 73, 31]
    arr4 = []
    arr5 = [68, 83, 6, 138, 20, 36, 71, 182, 108, 17, 96, 115, 194, 7, 150,
85, 177, 106, 53, 51, 66, 170, 111, 119,
            197, 141, 168, 37, 65, 78, 183, 189, 113, 110, 74, 49, 56, 45,
116, 97, 185, 24, 89, 64, 41, 193, 86, 54]
```

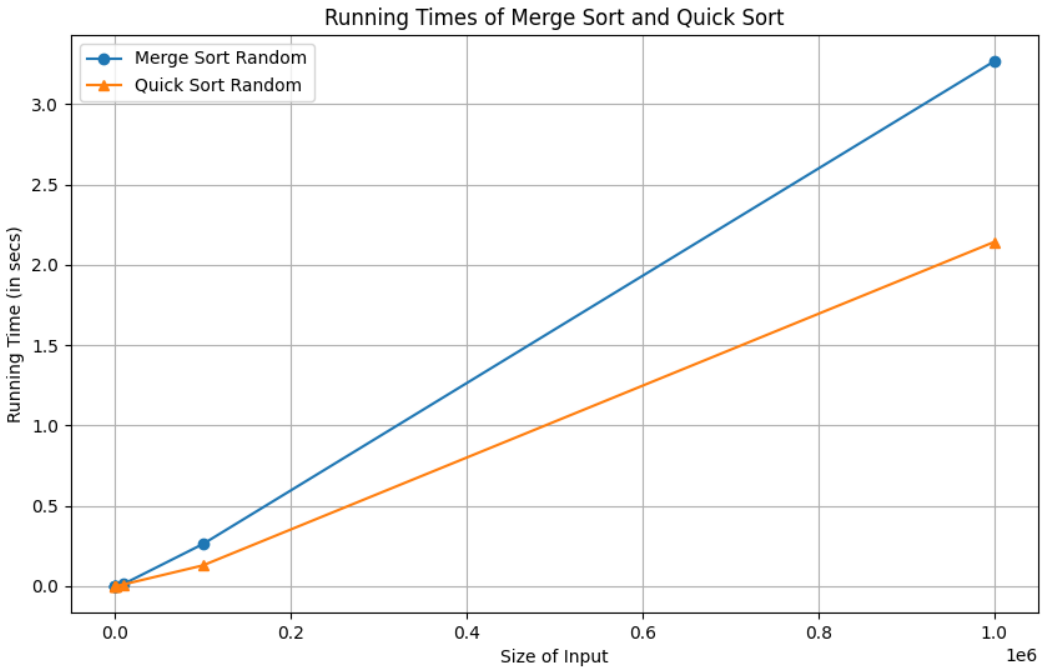
2. Plot the running times of the two algorithms against the *size* of the input (up to 10^7 numbers) when the input arrays are randomly sorted. Do the same when the inputs are already sorted. What functions best characterize the running times? Compare your results with what is expected from the theoretical analysis. Discuss the results and their implication.

1. **Inputs are randomly sorted**

The readings of the experiment are as follows :

Input Size	Random MergeSort	Random Quick Sort
1	9.5367431640625e-07	0.0
10	2.09808349609375e-05	1.2159347534179688e-05
100	0.00012302398681640625	6.103515625e-05
1000	0.001455068588256836	0.0009448528289794922
10000	0.014714956283569336	0.01159811019897461
100000	0.2625582218170166	0.12970399856567383
1000000	3.2676689624786377	2.142960786819458

If we plot the graph of the given points, we get –



Graph 1

Here, we make the following observations –

1. With respect to the graph above, we can see that Merge sort algorithm takes a longer time to sort a randomly sorted array than Quick sort algorithm.
2. Both the algorithms have a logarithmic growth in their running time.

What functions best categorize the running times?

1. Both the running times for both random merge sort and random quicksort are increasing as the input size (n) increases.
2. For merge sort of random input, we see that - The running times are in the order of 10^-6 to 10^-3 for the given input sizes. This suggests that the running time for random merge sort is roughly proportional to n log(n).

3. For quick sort of random input, we see that - The running times are in the order of 10^-6 to 10^-3 for the given input sizes. This suggests that the running time for random merge sort is roughly proportional to nlog(n).
4. As we see that the graphs indeed follow nlogn format. So, the function that best characterizes the running times here is f(n) = nlogn + c, where c can be any lower order constant terms.

Comparison with theoretical analysis

1. We know the time complexity of merge sort in all the 3 cases – best, worst and average case is O(nlogn) where n is the size of input.

2. We know from theoretical analysis that time complexity of quick sort is –
   a. Best/Average case – O(nlogn)
   b. Worst case – O(n^2)
3. As the plots have a logarithmic growth, this suggests that randomly sorted input is the best case of quicksort. It really doesn't matter which case this input falls for merge sort as it gives O(nlogn) for all cases.
4. This shows that the graph plotted aligns and the time complexity seen from the graph which is O(nlogn) aligns with the theoretical time complexity.

Results and their implications

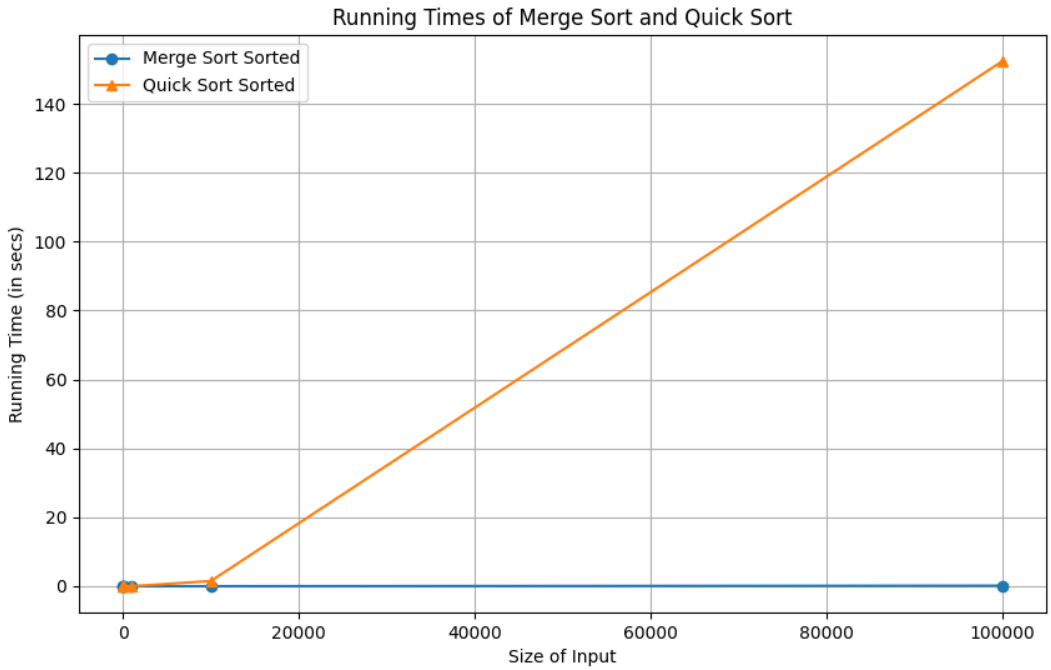
As we consider O(n log n) as a reasonable approximation for both Merge Sort and Quick Sort, with Quick Sort typically being faster in practice due to its lower constant factors and better cache behavior. The provided data seems to align with this expectation, with increasing runtimes as the input size grows, but specific constants and variations may depend on the exact implementation and hardware.

2. Inputs are already sorted

The readings are as follows –

Input Size	Sorted Merge Sort	Sorted Quick Sort
1	9.5367431640625e-07	0.0
10	2.002716064453125e-05	1.1205673217773438e-05
100	0.00011229515075683594	0.0001709461212158203
1000	0.0013208389282226562	0.016750812530517578
10000	0.01302194595336914	1.5162549018859863
100000	0.15620684623718262	152.37264204025269

When we plot the graph we get,



Graph 2

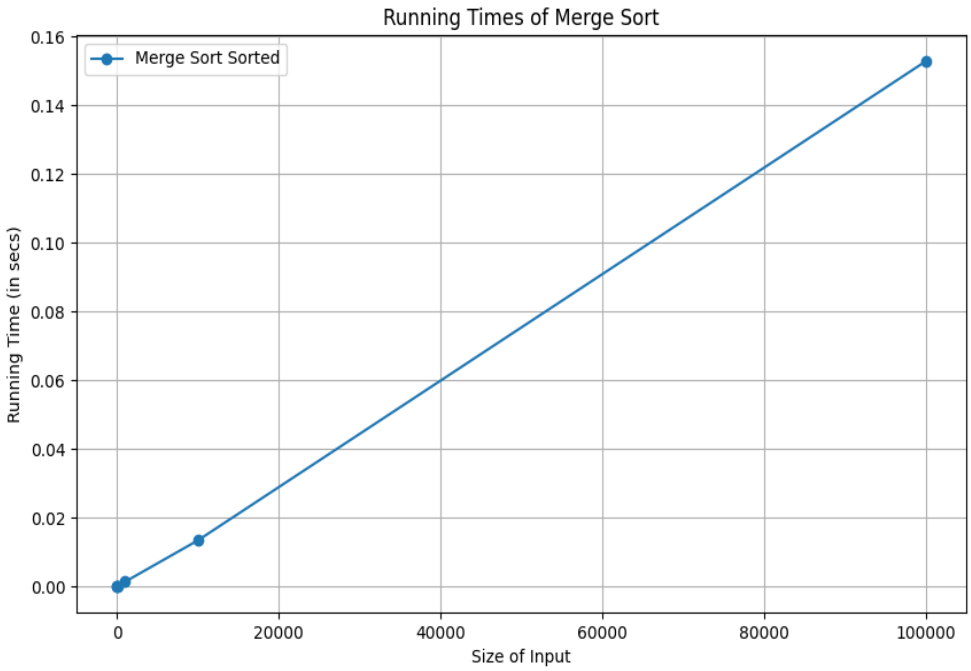
Here, we can make the following observations –

1. With respect to the graph above, we can see that Quick sort algorithm takes a lot longer time to sort a sorted array than Merge sort algorithm.
2. Both the algorithms have very different growth functions with respect to each other.

What functions best categorize the running times?

1. For quick sort of sorted input, we see from the graph that - The running times increase polynomial for given input sizes. This suggests that the running time for sorted quick sort is roughly proportional to n^2.

   For merge sort of sorted input, if we have a closer look (graph 3) we can see that - The running times are in the order of 10^-6 to 10^-3 for the given input sizes. This suggests that the running time for sorted merge sort is roughly proportional to n log(n).



Graph 3

2. As we see that both graphs follow different patterns (or growth), they will have different functions to characterizes the running times –
    a. In terms if quick sort, f(n) = n^2 + c1, where c can be any lower order constant terms is a function that base characterizes it.
    b. For merge sort, f(n) = nlog n + c2, where c can be any lower order constant terms is a function that base characterizes it.

Comparison with theoretical analysis

1. We know the time complexity of merge sort in all the 3 cases – best, worst and average case is O(nlogn) where n is the size of input.

2. We know from theoretical analysis that time complexity of quick sort is –
    a. Best/Average case – O(nlogn)
    b. Worst case – O(n^2)
    c. As the merge sort has a plot that has a logarithmic growth, this suggests that sorted input can be any case of merge sort. It really doesn't matter which case this input falls for mergesort as it gives O(nlogn) for all cases.
    d. As far as the quick sort is concerned, it gives the worst case time complexity when the input is sorted or the pivot is the least/largest item in the input array, which is **exactly** the case in the code. Hence, this scenario is the worst case input scenario in case of Quick sort.
    e. This shows that the graph plotted aligns and the time complexity seen from the graph which is O(nlogn) for mergesort and O(n^2) for quicksort (worst-case) aligns with the theoretical time complexity.

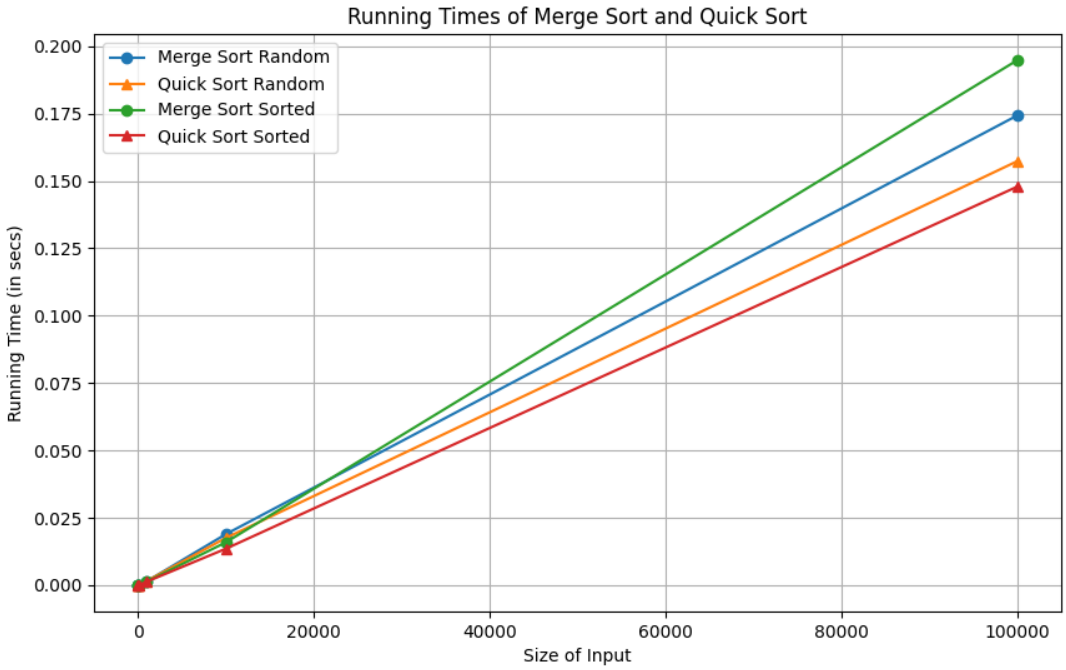
Results and their implications

As the code is using the 1st element as the pivot and the list is sorted, quick sort falls into its worst case scenario of O(n^2). This can be improved by making changes in the code where the partition chooses randomized pivot each time. Even if the randomized pivot partitions the array in such a way that an array of, lets say of length = 10 is split into 2 subarrays of 1 and 9 elements on each side, we can still obtain an average case complexity of O(nlogn). By the recursion method (question 3, average case complexity), this is proved.

Also, the fact that quicksort is an **in-place** sorting algorithm and merge sort **creates a new list** during the sorting process can have an impact on the running time of these algorithms in case of sorted input - Quicksort's performance can degrade on sorted input, especially if you always choose the first or last element as the pivot which results in O(n^2) time complexity. This is because the pivot selection may result in unbalanced partitions, causing quicksort to perform more comparisons and swaps than in the average case. This is also proved by my graph as Quick sort performs worse than merge sort.

But there is a solution with which we can improve the performance of quick sort which is by choosing a randomized pivot.

In addition to other graphs, randomized pivot graph is as shown below for all cases – for randomly sorted array and for sorted array for both merge sort and quick sort as shown below.



Graph 4

We can see from this graph, that quick sort sorted performance is improved and it is the fastest algorithm amongst all. Also, this graph depicts that the growth of quicksort for sorted input becomes proportional to nlog and hence, it depicts the best case performance even in terms of sorted input which was not the case earlier.

3. Express the running times (best, average, and worst case) of the quickSort in *O*-notation. Thoroughly justify your answers.

Let's assume that T(n) is the worst-case time complexity of quicksort for n integers. Let's analyze it by breaking down the time complexities of each process:

Divide Part: The time complexity of the divide part is the time complexity of the partition algorithm, which is O(n).

Conquer Part: We are recursively solving two subproblems of different sizes. The size of the subproblems depends on the choice of the pivot in the partition process! Suppose after the partition, i elements are in the left subarray (left of the pivot), and n - i - 1 elements are in the right subarray.

Size of the left subarray = i
Size of the right subarray = n - i - 1
Time complexity of the conquer part = Time complexity of sorting the left subarray + Time complexity of sorting the right subarray = T(i) + T(n - i - 1)

Combine Part: As mentioned above, there is no operation in this part of the quick sort.

So time complexity of the combine part = O(1)

For calculating the overall time complexity T(n), we need to add the time complexities of the divide, conquer, and combine part:

T(n) = O(n) + T(i) + T(n - i - 1) + O(1)
= O(n) + T(i) + T(n - i - 1)
= T(i) + T(n - i - 1) + cn

So now the recurrence relation of the quick sort will be –
T(n) = c, if n = 1
T(n) = T(i) + T(n — i — 1) + cn, if n > 1

Worst-case analysis using Recursion Method

The worst-case scenario will occur in a scenario when the partition process chooses the largest or smallest element as the pivot every time(as we can see from graph 2). This will result in the partition being highly unbalanced, with one subarray containing n - 1 elements and the other subarray containing 0 elements.

When we always choose the rightmost (or left most) element as the pivot, the worst-case scenario will arise when the array is already sorted in either increasing or decreasing order(graph 2). In this case, each recursive call will create an unbalanced partition.

For calculating the time complexity in the worst case, we put i = n — 1 in the above equation of T(n):

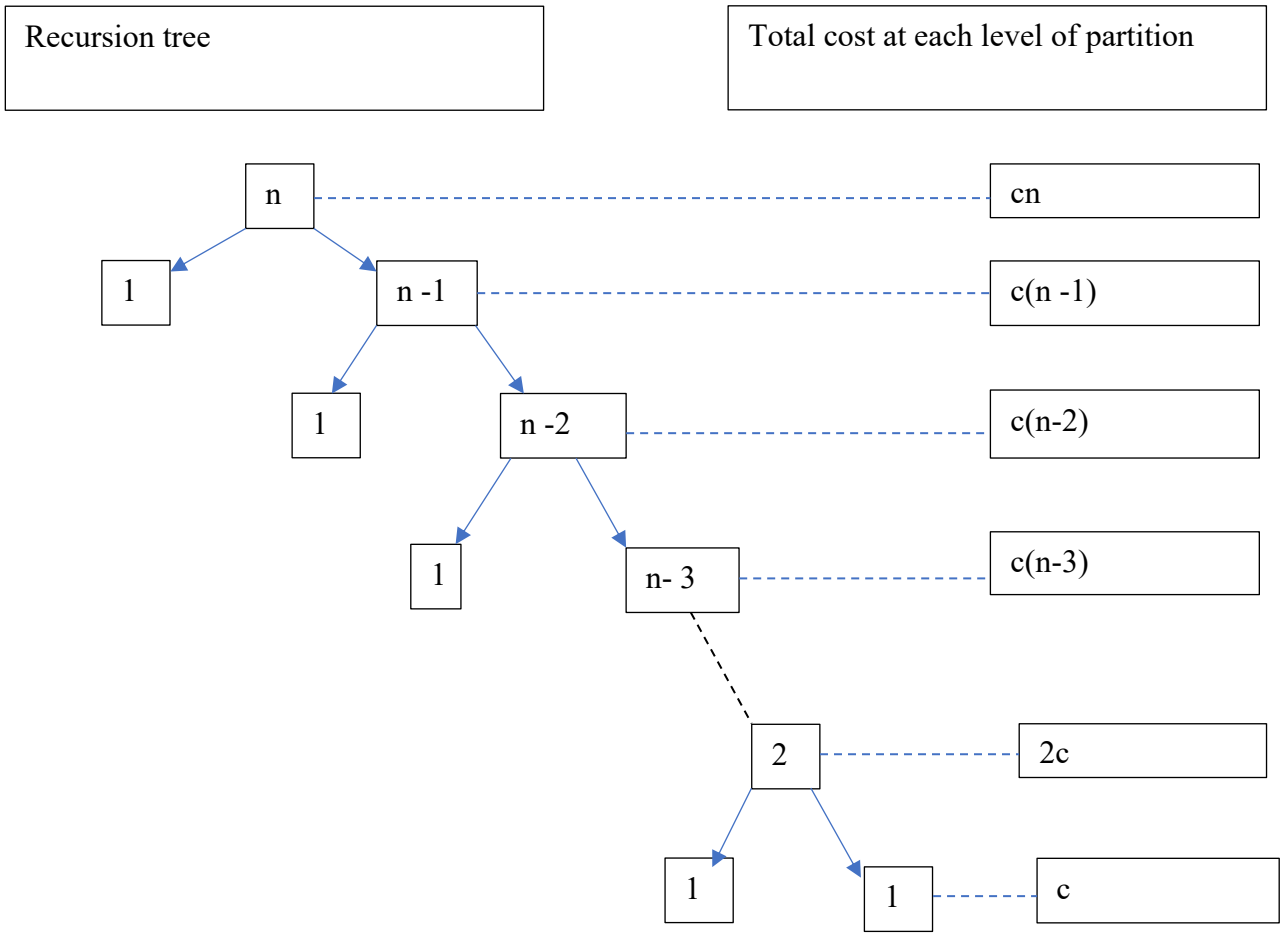
T(n) = T(n — 1) + T(0) + cn

= T(n — 1) + cn

We know the recurrence relation is - T(n) = T(n — 1) + cn.
Which means that if we partition the given array, it will result in T(n -1) items in one subarray and cn (or constant) which is just 1 element in the other subarray.

Hence to calculate the cost of work done at each partition, we take into account only the heavier subarray.

Consider the recursion tree diagram of quick sort in worst case –



Here, if we consider the total time taken as c, the subarrays keep on subtracting one element from themselves and go deeper till it reaches 1.

To find the total time complexity taken we will add the total partitioning cost for each level.
Hence, we get
T(n) = cn + c(n−1) + c(n−2) +⋯+ 2c + c
= c (n + n−1 + n−2 + ⋯+ 2 + 1)

As this is sum of n natural numbers, T(n) will be
T(n) = c[n(n+1)/2]
= O(n^2).

Therefore, the worst case time complexity of Quicksort is O(n^2).

Best-case analysis of quick sort using recursion method

The best-case scenario of quick sort will occur when partition process always picks the median item of the array as the pivot. In other words, this will result in balanced partition, where both sub-arrays are approx. n/2 size each and equal in size. The best case also occurs when we randomize the selection of the pivot(as depicted in graph 4).

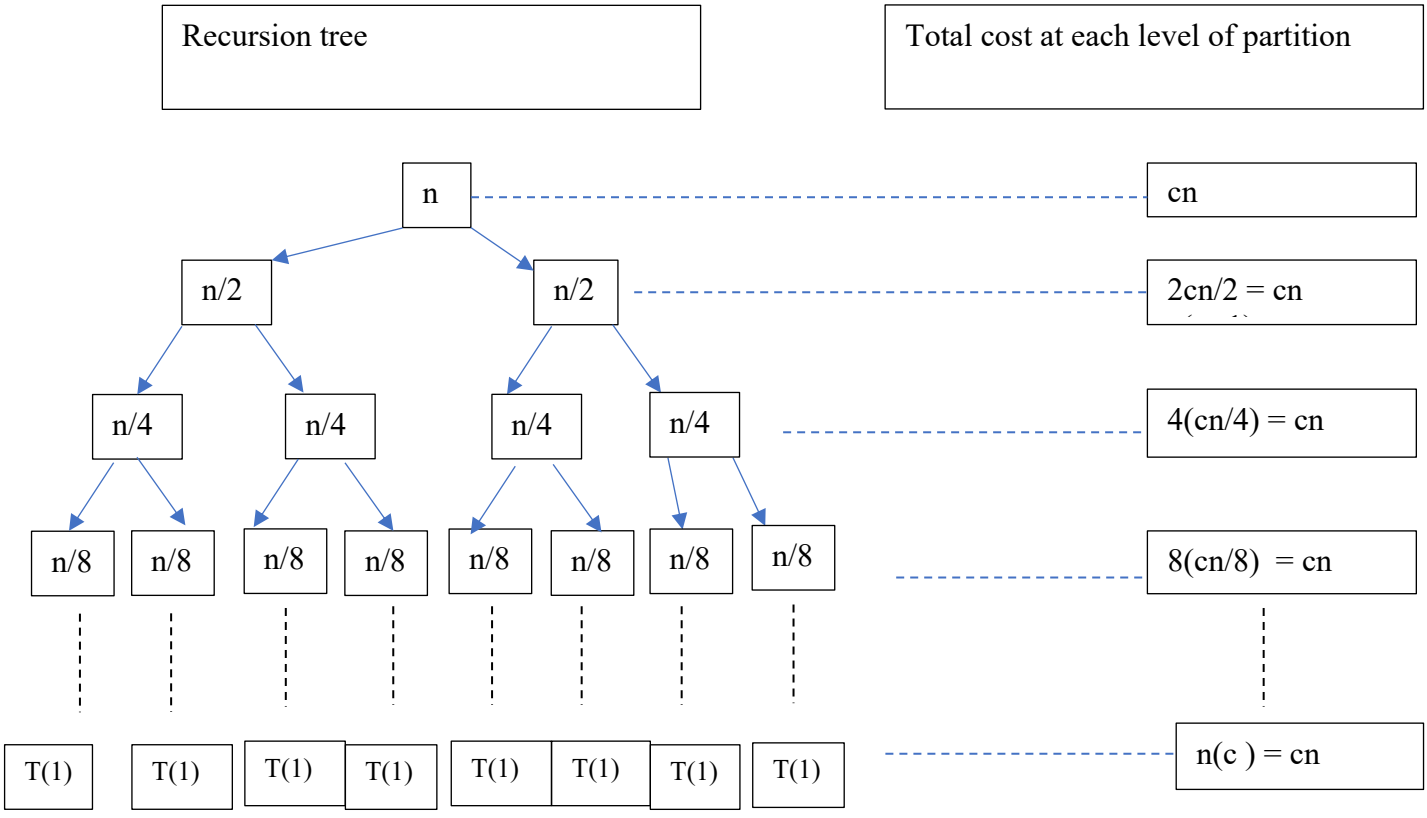
Lets assume, the scenario of balanced partitioning will arise in each recursive call. Now, for calculating the time complexity in the best case, we put i = n/2 in the above formula of T(n).

T(n) = T(n/2) + T(n – 1 – n/2) + cn
= T(n/2) + T(n/2 – 1) + cn
= 2T(n/2) + cn

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

Using Recursion Method

The recursion tree diagram of quick sort in best case will be as follows -



Here we can see that if n is getting equally divided into 2 halves every time then the no. of levels = log2(n+1)
So the total time complexity = No. of levels * cost of each level
= Log2(n+1) * cn
= O(nlogn)

Therefore, the best case time complexity of quick sort is O(nlogn)

Average case analysis of Quick Sort using recursion method.

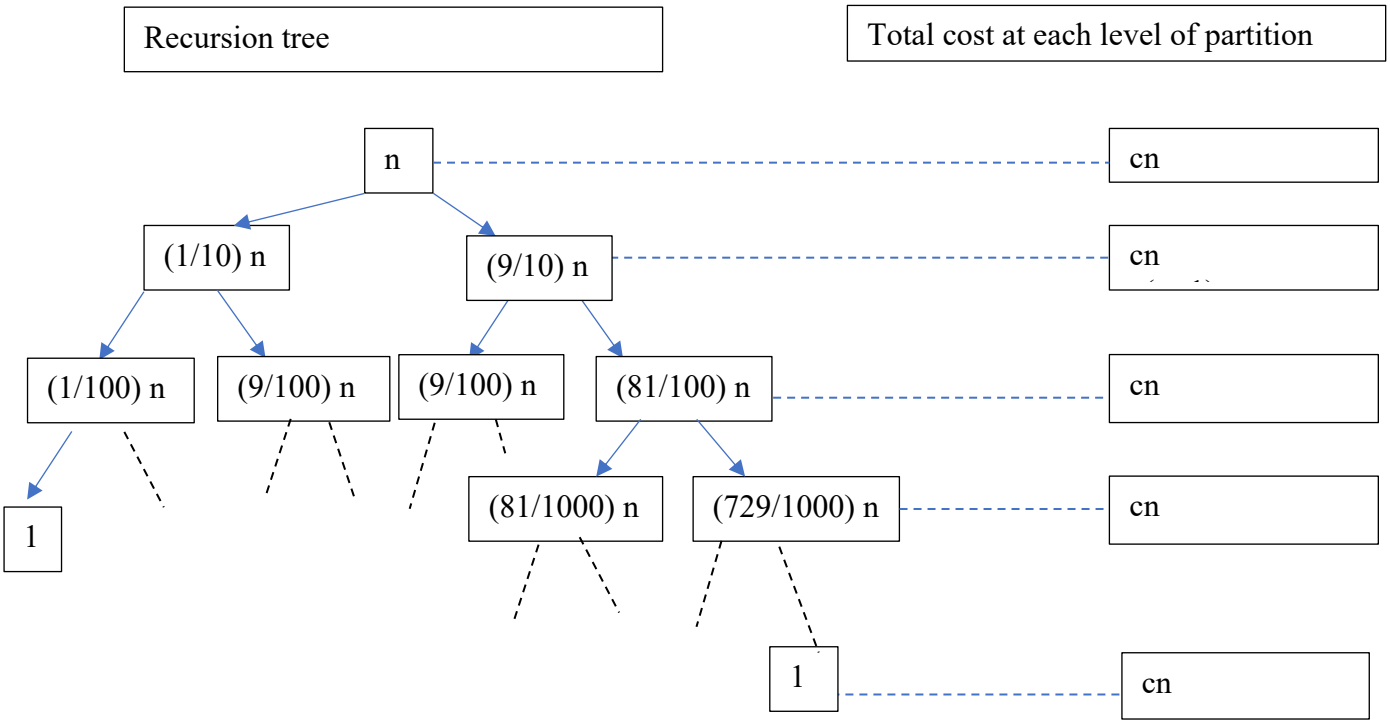
While considering the average case of recursion on random input, partitioning is highly unlikely to happen in the same way at each level of recursion. So, the behavior of quicksort algorithm will depend on the relative order of values in the input.

Here some of the subarrays will be reasonably well balanced and some will be unbalanced. So, the partition process will generate a mix of balanced partitions and unbalanced partition in the average case. These good and bad splits will be distributed randomly throughout the recursion tree.
The average case also occurs when we randomize the selection of the pivot(as depicted in graph 4).

For a better understanding of the analysis, we consider that balanced and unbalanced partition will appear at alternate levels of the tree.

Let's assume, at the root, there is a balanced partition, and at the next level, the partition generates an unbalanced one. The cost of the partition process will be O(n) at both levels. Therefore, the combined partitioning cost of the bad split followed by the balanced one is O(n).

Altogether, this is equivalent to a single level of partitioning, which resembles the scenario of a balanced partition. As a result, the height of the recursion tree will be O(log n), the combined cost of each level will be O(n), and the average-case running time of quicksort will be O(n log n).
Assume the recursion tree as follows –



Some of the observations are:

1. The left subtree is decreasing fast with a factor of 1/10. So the depth of left subtree is equal to log10(n).
2. The right subtree is decreasing slowly with a factor of 9/10. So the depth of right subtree is equal to log10/9(n).So, log10/9(n) = O(logn).

Hence, at each level of recursion, the cost of partition is at most cn. After doing the sum of cost at each level of recursion tree, quick sort cost is O(nlogn).

To conclude we can say that in general(or on average) any split of constant proportionality produces a recursion tree of depth O(logn), where cost at each level is O(n). So time complexity is O(nlogn) whenever the split has constant proportionality.

Hence, the average case time complexity of quick sort is O(nlogn)