DEPARTMENT OF COMPUTER ENGINEERING

EXPERIMENT NO 4

Aim: - To implement Quick Sort Algorithm using Divide and Conquer and find its complexity.

Lab Outcomes: CSL401.1: Ability to implement the algorithm using divide and conquer approach and also analyse the complexity of various sorting algorithms.

Theory: -

The basic version of the 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.

Good points

- 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.

Bad Points

- 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.

Quick sort works by partitioning a given array $A[p \ldots r]$ into two non-empty sub array $A[p \ldots q]$ and $A[q+1 \ldots r]$ such that every key in $A[p \ldots q]$ is less than or equal to every key in $A[q+1 \ldots r]$. Then the two sub arrays are sorted by recursive calls to Quick sort. The exact position of the partition depends on the given array and index q is computed as a part of the partitioning procedure.

Quick Sort

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Note that to sort entire array, the initial call Quick Sort (A, 1, length[A])

As a first step, Quick Sort chooses as pivot one of the items in the array to be sorted. Then array is then partitioned on either side of the pivot. Elements that are less than or equal to pivot will move toward the left and elements that are greater than or equal to pivot will move toward the right.

Partitioning the Array

Partitioning procedure rearranges the sub arrays in-place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Partition selects the first key, $A[p]$ as a pivot key about which the array will partitioned:

Keys $\leq A[p]$ will be moved towards the left .
Keys $\geq A[p]$ will be moved towards the right.

The running time of the partition procedure is $\Theta(n)$ where $n = r - p + 1$ which is the number of keys in the array.

Another argument that running time of PARTITION on a sub array of size $\Theta(n)$ is as follows: Pointer i and pointer j start at each end and move towards each other, conveying somewhere in the middle. The total number of times that i can be incremented and j can be decremented is therefore $O(n)$. Associated with each increment or decrement there are O(1) comparisons and swaps. Hence, the total time is $O(n)$.

Quicksort implementation
Suppose the function call QUICKSORT(A[], p, r) sorts the entire array where p and r are the left and right ends.
Divide: We define PARTITION (A, p, r) function inside QUICKSORT () method to divide array around a pivot. By the end of this method, we return the index of the pivot i.e. q = PARTITION (A, p, r)
Conquer: We recursively call QUICKSORT (A, p, q-1) to sort the left half and recursively call QUICKSORT (A, q+1, r) to sort the right half.
Base case: If we find p≥r during the recursive calls, the sub-array is either empty or has one value left. Our recursion should not go further and return from here.

Partition algorithm quick sort
Now, how can we divide array around pivot so that elements in the left half are less than the pivot and elements in the right half are greater than the pivot? Is it feasible to do it in place? Think!

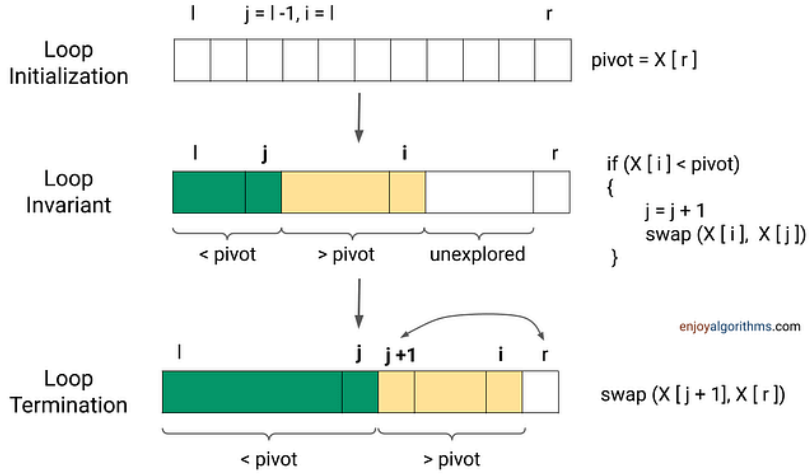
DEPARTMENT OF COMPUTER ENGINEERING

Suppose we pick pivot = A[r] and scan the array from left to right. Whenever we find some A[j] less than the pivot, we place it in the starting part of the array. Otherwise, we ignore A[j]. For this, we need two pointers to track two parts: The pointer j to track the current index and the pointer i to track the starting subarray (from 0 to i) where values are less than pivot. At the start i = p -1 and j = l.

Now we traverse array from j = l to r - 1. When A[j] < pivot, we increment i and swap A[j] with X[i]. Otherwise, we ignore X[j] and move forward to the next element. This loop will terminate when j = r.

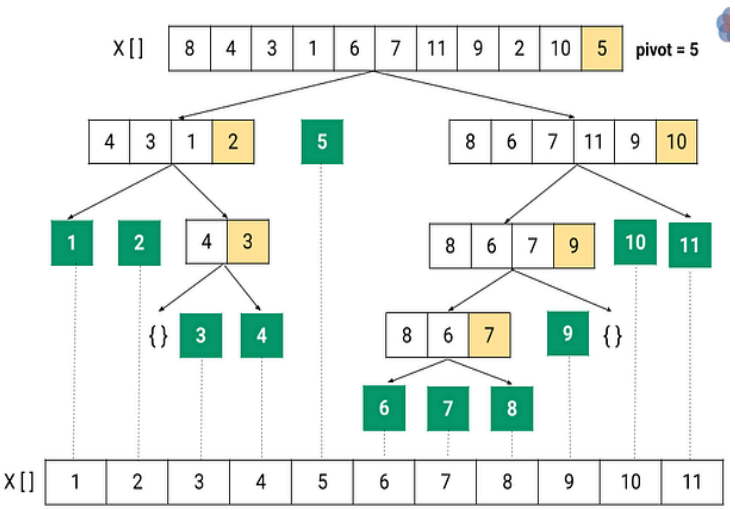
After the loop, values in subarray A[0…i] will be less than the pivot, and values in subarray A[i…r - 1] will be greater than the pivot. So we swap the pivot (A[r]) with A[i + 1] to complete the partition process because the correct position of the pivot will be index i + 1.

By the end of the partition process, we return index i + 1.



Analysis of the partition algorithm

We are running a single loop and doing constant operations at each iteration. So, time complexity = O(n). We are using constant extra space, so space complexity = O(1).



Quicksort algorithm visualisation on array

Analysis of the quicksort

We first need to define the recurrence relation to analyse the recursive function. Suppose T(n) is the time complexity of quicksort for input size n. To get the expression for T(n), we add the time complexities of each step in the above recursive code:

Divide step: The time complexity of this step is equal to the time complexity of the partition algorithm i.e. O(n).

Conquer step: We solve two subproblems recursively, where subproblem size will depend on the value of the pivot during partition algorithm. Suppose i elements are on the left of the pivot and n — i — 1 elements are on the right of the pivot after partition.

- Input size of left subarray = i
- Input size of right subarray = n — i — 1

Conquer step time complexity = Time complexity to sort left subarray recursively + Time complexity to sort right subarray recursively = T(i) + T(n — i — 1)

Combine step: This is a trivial step and there is no operation in the combine part of quick sort. So time complexity of combine step = O(1).

T(n) = O(n) + T(i) + T(n — i — 1) + O(1)

= T(i) + T(n — i — 1) + O(n)

= T(i) + T(n — i — 1) + cn

Recurrence relation of the quick sort:

- T(n) = c, if n = 1
- T(n) = T(i) + T(n — i — 1) + cn, if n > 1

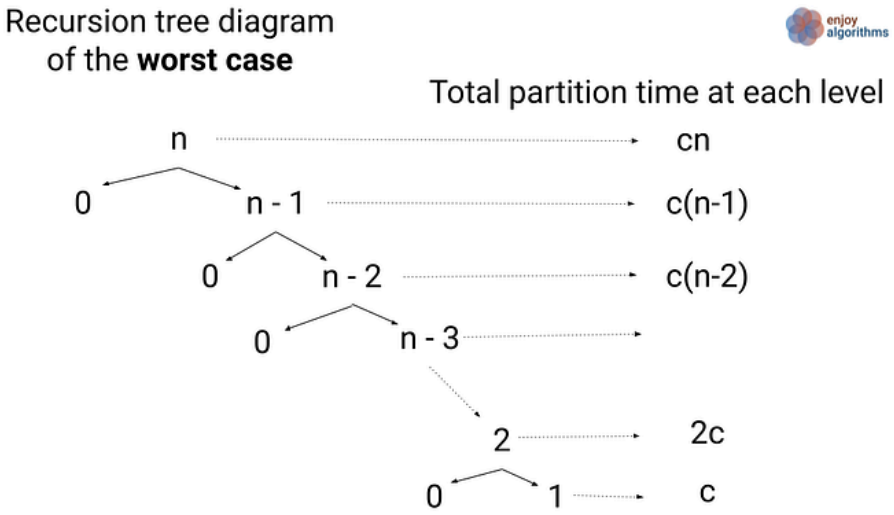
Worst-case time complexity analysis of the quick sort

Quick sort worst-case occurs when pivot is always the largest or smallest element during each call of partition algorithm. In such a situation, partition process is highly unbalanced, where one

subarray has n — 1 element and the other subarray is empty. Note: Worst case always occurs in the case of sorted or revere sorted array. Think!

So, for calculating quick sort 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

Analysis using the recursion tree method



In this method, we draw the recursion tree and sum the partitioning cost for each level => cn + c(n−1) + c(n−2) +⋯+ 2c + c = c (n + n−1 + n−2 + ⋯+ 2 + 1) = c(n(n+1)/2) = $O(n^2)$. So worst case time complexity of quick sort is $O(n^2)$.

Best case time complexity analysis of the quick sort

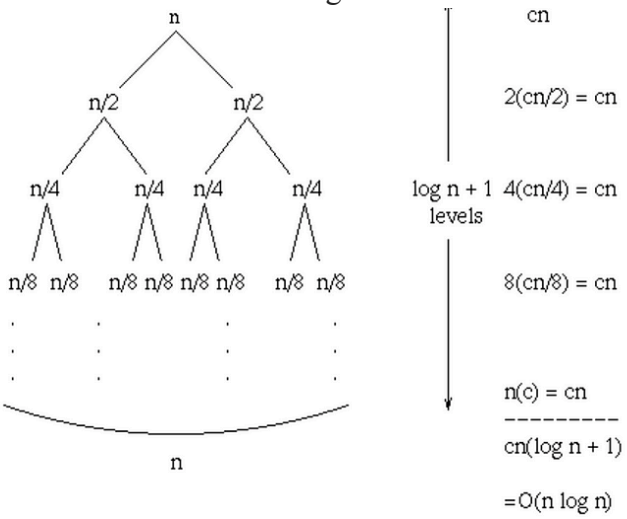
Quick sort best case occurs when the pivot is always the median element during each call of the partition algorithm. In such a situation, the partition process is balanced and the size of each sub-problems is approx. n/2 size.

So, for calculating quick sort time complexity in the best case, we put i = n/2 in the above equation of T(n) => T(n) = T(n — 1) + T(0) + cn = T(n — 1) + cn

So for the time complexity in the best case, we put i = n/2 in the above equation of T(n) => T(n) = T(n/2) + T(n — 1 — n/2) + cn = T(n/2) + T(n/2 — 1) + cn ~ 2 T(n/2)+ cn

This is similar to the recurrence relation of merge sort. So we can solve this using both the recursion tree method and the master theorem. In other words, the best-case time complexity of quick sort is equal to the time complexity of merge sort which is O(nlogn).

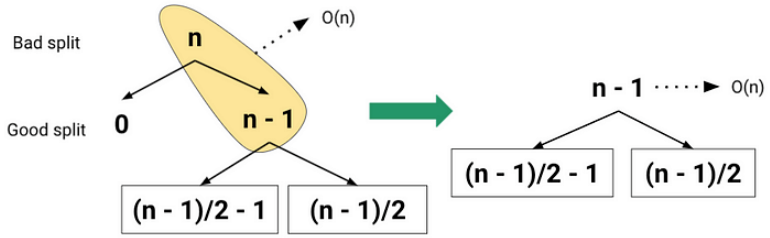
Here is the recursion tree diagram for the best case of quick sort.



Average case time complexity analysis of the quick sort

When we select pivot randomly during the partition process, partitioning is highly unlikely to happen in the same way at every level, i.e. some of the splits will be close to a balanced scenario, and some will be close to an unbalanced scenario. In such a practical scenario, an average case analysis of quick sort will give us a better picture. Note: Here, we assume that the probability of choosing each element as the pivot is equally likely.

In the average case, the partition process will generate a combination of good (balanced partition) and bad (unbalanced partition) splits, where good and bad splits will be distributed randomly in the recursion tree.

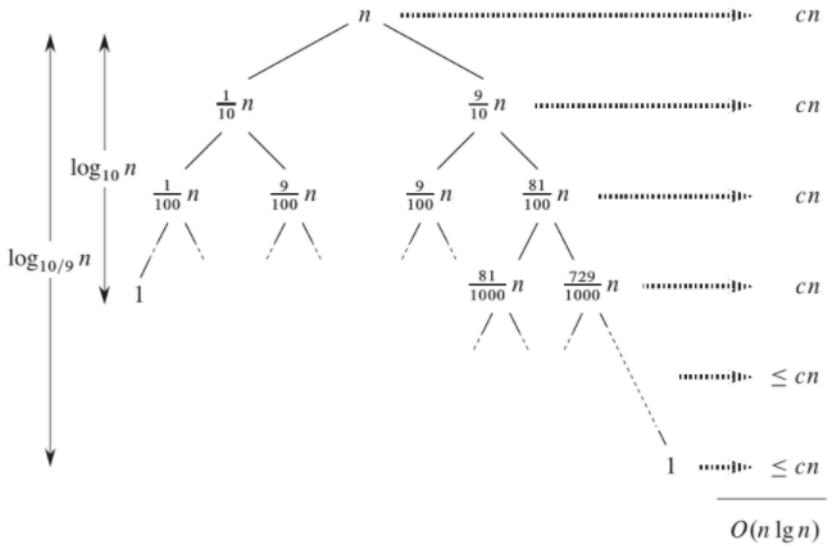


Suppose at the first level of the recursion tree, the partition process creates a good split, and at the next level, the partition generates a bad split. So the combined cost of the partition process at both levels will be O(n). In other words, the combination of an unbalanced partition followed by a balanced partition will take O(n) partitioning time. This looks similar to the scenario of a balanced partition!

So the average-case time complexity of quicksort will be closer to the best-case time complexity O(nlogn). For better understanding, suppose the partition algorithm always generates a partially unbalanced spit in the ratio of 9 to 1.

Recurrence relation: T(n) = T(9n/10) + T(n/10) + cn.

The following diagram shows the recursion tree for this recurrence.



We can notice the following things from the above diagram:

- The left part of the recursion tree is decreasing fast with a factor of 1/10 and the right part is decreasing slowly with a factor of 9/10. So in the worst case, the maximum depth of the recursion tree will be equal to log10/9(n), which is O(logn).
- The partitioning cost is at most cn at every level of the recursion tree. After doing level by level sum, the total cost of the quick sort will be cn*O(logn) = O(nlogn). So for an unbalanced split of 9-to-1, quicksort works in O(nlogn) time complexity.
- In general, time complexity will be O(nlogn) whenever split has constant proportionality. For example, 99-to-1 split will also produce O(nlogn) time complexity for quick sort.

Conclusion: - Thus we have implemented a recursive quick sort using divide and conquer approach and analysed its complexities.