CSEN3001:
DESIGN AND ANALYSIS OF ALGORITHMS

UNIT-I: INTRODUCTION TO ALGORITHMS
QuickSort

Divide and conquer – general idea

- **Divide a problem into subprograms of the same kind**
- **Solve subprograms using the same approach**
- **Combine partial solution (if necessary)**

Quick sort


Pivot Left Right
54 26 93 17 77 31 44 55 20
Initial List
Pivot Left Right
54 26 93 17 77 31 44 55 20
Compared left with pivot till arr[left] < pivot and right with pivot till arr[right] > pivot.
Pivot Left Right
54 26 20 17 77 31 44 55 93
Exchanged left and right elements
Pivot Left Right
54 26 20 17 77 31 44 55 93
Compared left with pivot till arr[left] < pivot and right with pivot till arr[right] > pivot.
Pivot Left Right
54 26 20 17 44 31 77 55 93
Exchanged left and right elements
Right Left
54 26 20 17 44 31 77 55 93
Right and left crossed over so stop. The right index is the split point and is the place where pivot element should be placed and we need to swap pivot with element at right.
31 26 20 17 44 54 77 55 93
List after swapping

Algorithm: Quick Sort

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], …., a[q] which reside in the global array a[1:n] into ascending order; //a[n+1] is considered to be defined and must be >= all the  elements in a[1:n]
{
        if (p<q) then
       {
        j := partition(a,p, q+1);  // divide P into two subproblems; j is partitioning element
        // Solve the subproblems
        QuickSort(p, j-1)
        QuickSort(j+1, q)
    // There is no need for combining solutions
   }  end of if
 } // end of algorithm
```

Algorithm: partition and interchange

Algorithm partition(a, m, p)

```
// within a[m], ……a[p-1] the elements are rearranged in such a
//manner that if initially t=a[m], then after completion a[q]=t for
//some q between m and p-1,a[k]<=t for m<=k<q and a[k] >=t for
//q<k,p.  q is returned. Set a[p]=∞
{
        v:=a[m]; i:=m; j:=p;
        do {  do   i:=i+1;  while (a[i] < v);
                do   j:=j-1;   while a[j] > v);
                if  (i<j) then interchange(a,i,j);
          } while (i<j)
  a[m]:=a[j]; a[j]:= v; return j;
}
```

Algorithm interchange(a,i,j)

```
// Exchange a[i] with a[j]
{
        p:=a[i];
        a[i]:=a[j];
        a[j]:=p;
}
```

Time Complexity: worst case

- Consider the element comparisons only C(n)
- Assume n elements are distinct and the input distribution is such that the partitioning element v=a[m] in the call Partition(a, m, p) has an equal probability of being the i^{th} smallest element $1 \leq i \leq p-m$ in a[m:p-1]
- Worst Case Complexity: $C_W(n)$ of C(n)
- Number of element comparisons in each call of partition is at most : p- m +1
- Let r be the total number of elements in all the calls to Partition at any recursion level.
- At level one only one call : Partition(a, 1, n+1) is made and r=n;
- At level two, at most, two calls are made r=n-1, and so on.
- At each level of recursion, O(n) element comparisons are made.
- $C_W(n)$ = sum of r, varies from 2 to n or $O(n^2)$
- Therefore, Quick sort uses $\Omega(n^2)$ comparisons

Time Complexity: average case

- Average Case: $C_A(n)$ of $C(n)$
- The partitioning element r has an equal probability of being the i^{th} smallest element $1 \leq i \leq p-m$ in a[m:p-1]
- Hence the two sub arrays remaining to be sorted are a[m:j] and a[j+1:p-1] with probability 1/(p-m), $m \leq j < p$
- From this, obtain the following recurrence:
- $C_A(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n} C_A\,(k-1) + C_A(n-k))$ ------ Eq. 1
- The number of element comparisons required by Partition on its first call is n+1
- $C_A(0)=C_A(1)=0$
- Multiplying Eq.1 by n
- $nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots . + C_A(n-1)$ --- Eq. 2

Time Complexity: average case

Replacing n by n-1 in Eq.2

$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + C_A(1) + \cdots .. + C_A(n-2)]$ ---- Eq.3

Eq. 3 – Eq.2 :

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

Or

$$nC_A(n) = (n+1)C_A(n-1) + 2n$$

Multiplying by 1/(n*(n+1))

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-1)}{n} + \frac{2}{n+1}$$

Repeatedly using this equation to substitute for CA(n-1), CA(n-2) … we get:

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

Time Complexity: average case

$$\frac{C_A(n)}{n+1} = \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \ldots. \quad = \quad 2\sum_{k=3}^{n+1} \frac{1}{k}$$

$$\sum_{k=3}^{n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{x} dx = log_e(n+1) - log_e 2$$

$$C_A(n) \leq 2(n+1)[log_e(n+2) - log_e 2 = O(n \log n)$$

The average time is O(n log n)

Space requirement: Stack Space

The maximum depth of recursion in the worst case is n-1. The amount of stack space needed can be reduced to O(log n) by using an iterative version of quicksort in which the smaller of the two subarrays, a[p:j-1] and a[j+1:q], is always sorted first.

Maximum stack space needed:

$$\leq \begin{cases} 2 + S(\lfloor \frac{n-1}{2} \rfloor & n > 1 \\ 0 & n \leq 1 \end{cases}$$

Algorithm Selection – To find the kth smallest element

```
Algorithm Select1(a,n,k)
// Selects the kth smallest element in a[1:n] and places it in the kth position of a[]. The
//remaining elements are rearranged such that a[m]<=a[k] for 1<=m<k and a[m]>=a[k] for
// k<m<=n.
{ low:=1; up:=n+1; a[n+1]= infinity;
  do {
        // each time the loop is entered, 1<=low<=k<=up<=n+1.
        j=Partition(a, low, up); // j is such that a[j] is the jth smallest value in a[].
        if (k==j) then return a[j];
        else if (k<j) then up=j; // j is the new upper limit
             else low=j+1;       // j+1 is the new lower limit
        } while(true);
}
```

THANK YOU