ASSIGNMENT-1

Write algorithm for following divide & Conquer applications

(i) Merge Sort:
(ii) Quick Sort
(iii) Binary Search
(iv) Strassen's Matrix Multiplication

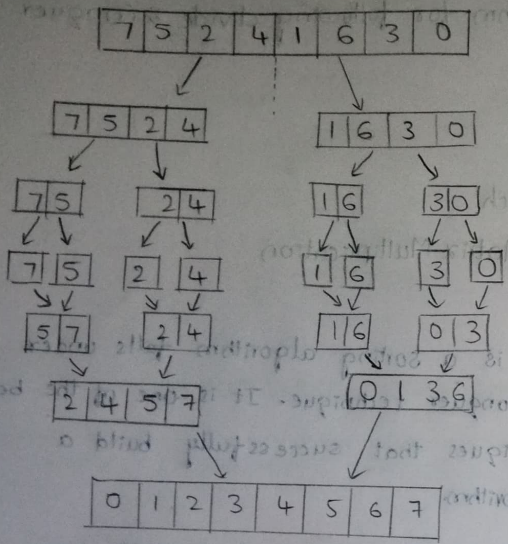
(i) Merge Sort:

Merge Sort is a sorting algorithm falls under Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

The merge sort is a comparision sort and has an algorithmic complexity of O(nlogn). Elementary implementations of merge sort make use of two arrays - one for each half of the data set.

Marginally faster than the heap sort for large data set. It always does less number of comparisions than Quick sort.



Algorithm for merge sort

* Suppose we have a array A whose starting index is P and ending index is r
* let q be the mid of the array in between p, r
* We devide array into A[P,q] and A[q+1,r]
* We sort theses array and merge them

1. If P<r
2. Then q → (P+r)/2
3. MERGE SORT (A, P, q)
4. MERGE SORT (A, q+1, r)
5. MERGE (A, P, q, r)

Functions: MERGE (A, P, q, r)

1. n1 = q - P + 1
2. n2 = r - P
3. Create arrays [1,.....n1+1] and R[1.....n2+1]
4. for i → 1 to n1
5. do [i] → A[P+i-1]
6. for j → 1 to n2
7. do R[i] → A[q+j]
8. L[n1+1] → ∞
9. R[n2+1] → ∞
10. I → 1
11. J → 1
12. For k → P to r
13. DO if L[i] ≤ R[j]
14. then A[K] → L[i]
15. i → i+1
16. else A[K] → R[j]
17. j → j+1

Computing time for merge sort:

The time for the merging operation is proportional to n, then computing time for merge sort is described by using recurrence relation

$$T(n) = \begin{cases} a & \text{if } n=1 \\ 2T(n/2) + cn & \text{if } n>1 \end{cases}$$

Here $c, a \to$ constants

If n is power of 2, $n = 2^k$

Form recurrence relation i.e., $n = n/2$

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

$2[2T(n/4) + cn/2] + cn$

$4T[n/4] + 2cn$

$2^2 T[n/4] + 2cn$

$2^3 T(n/8) + 3cn$ for $n = n/4$

$2^4 T(n/16) + 4cn$ for $n = n/8$

$2^k + T(1) + k(n)$

$an + cn(\log n)$

By representing it by in the form of Asymptotic notation O is $T(n) = O(n\log n)$

(ii) Quick Sort

Quick sort is an algorithm based on divide and conquer paradigm. that selects a pivot element and reorders the given list in such a way that all elements smaller to it are on one side and those bigger than it are on the other. Then the sublists are recursively sorted until the list gets completely sorted. The time complexity of this algorithm is $O(n\log n)$.

Auxiliary space used in the average case for implementing recursive function calls is $O(\log n)$ and hence proves to be a bit space costly, especially when it comes to large data sets.

Its worst case has a time complexity of $O(n^2)$ which can prove very fatal for large data sets.

Algorithm for QuickSort

```
Algorithm quicksort (a,low,high)
{
  if (high > low) then {
    m = partition (a,low, high);
    if (low<m) then quick (a,low,m);
    if (m+1 < high) then quick (a,m+1, high);
  }
}
Algorithm partition (a, low, high)
{
  i= low , j=high ;
  mid = (low+ high)/2 ;
  pivot = a[mid];
```

```
while (i<=j) do { while (a[i] = pivot)
    i++;
  while( a[j] > pivot)
    j--;
  if(i<=j) {
     temp = a[i];
     a[i] = a[j];
     a[j] = temp;
     i++;
     j--;
  }
}
return j;
}
```

Time Complexity:

Best Case : $O(\log(n))$

Average Case : $O(n\log n)$

Worst Case : $O(n^2)$

(iii) Binary Search

* This algorithm finds the position of a specified input value (the search "key") with an array sorted by key value.
* In each step, the algorithm compares the search key value with the key value of the middle element of the array.
* If key matches, then matching element has been found and its index, or position its returned.
* Otherwise, if the search key is less than the middle element's key then algorithm repeats its action on the sub array to the left of the middle element or if the search key is greater than then the algorithm repeats on sub array to the right of the middle element.
* If search element is less than the minimum position element or greater than maximum position element then this algorithm returns not found.

Algorithm for binary Search (recursive)

```
Algorithm binary-search (A, key, imin, imax)
{
  if (imax<imin) then
    return "array is empty";
```

if (key < imin || k > imax) then

return "element not in array list"

else

{

imid = (imin + imax)/2;

if (A[imid] > key) then

return binary_search (A, key, imin, imid-1);

else if (A[imid] < key) then

return binary_search (A, key, imid+1, imax);

else

return mid;

}

}

For successful Search

worst case → $O(\log n)$ or $\Theta(\log n)$

Average case → $O(\log n)$ or $\Theta(\log n)$

Best case → $\Theta(1)$ or $O(1)$

unsuccessful Search

$O(\log n)$ for all cases.

(ii) Strassen's Matix Multiplication

In matrix multiplication considering x, y, z matrices $z = x \times y$. Using Naive method, two matrices (x and y) can be multiplied if the order of these matrices are $p \times q$ and $q \times r$

Algorithm for Naive Method

```
for i=1 to p do
    for j=1 to r do
        z[i,j] := 0
        for k=1 to q do
            z[i,j] = z[i,j] + x[i,k] x y[k,j]
```

Here we assume that integer operations take $O(1)$ time. There are three for loops in this algorithm and one is nested in other. Hence, the algorithm takes $O(n^3)$ time to execute.

In this context, using Strassen's Matrix Multiplication algorithm, the time consumption can be improved a little bit.

Strassen's Matrix Multiplication can be performed only on square matrices where n is a power of 2, Order of both of the matrices are $n \times n$.

Divide X, Y and Z into four $(n/2) \times (n/2)$ matrices are represented below -

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix}, \quad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \quad Y = \begin{bmatrix} E & F \\ G \times H \end{bmatrix}$$

Using Strassen's Algorithm compute the following:

$$M_1 = (A+C) \times (E+F)$$

$$M_2 = (B+D) \times (G+H)$$

$$M_3 = (A-D) \times (E+H)$$

$$M_4 = A \times (F-H)$$

$$M_5 = (C+D) \times E$$

$$M_6 = (A+B) \times H$$

$$M_7 = D \times (G-E)$$

Then $I = M_2 + M_3 - M_6 - M_7$

$J = M_4 + M_6$

$K = M_5 + M_7$

$L = M_1 - M_3 - M_4 - M_5$

Analysis

$$T(n) = \begin{cases} c & \text{if } n=1 \\ 7 \times T(n/2) + d \times n^2 & \text{otherwise} \end{cases}$$

where, c and d are constants.

Using this recurrence relation $T(n) = O(n^{\log 7})$

Hence, complexity of Strassen's matrix multiplication algorithm is $O(n^{\log 7})$