Divide and Conquer Strategy

Divide and Conquer is a design approach that can be applied to several set of problems

It divides the problem into several subproblems that are similar to the original problem but smaller in size, solves the subproblems recursively (again divides subproblem into smaller subproblems) and then combines these solutions to create a solution to the original problem

The three steps used are

- **Divide** – the problem into subproblems

- **Conquer** – small subproblems are solved by using some efficient method while for larger subproblems divide and conquer approach is recursively applied
- **Combine** – the solutions of the subproblems is combined to get the solution of the original problem

A design strategy can be applied to a variety of problems

A control abstraction is a procedure whose flow of control is well defined but operations are themselves procedures which are not precisely defined but can be replaced by well defined procedures for a particular problem

```
Algorithm DivideAndConquer(P)
{
 if Small(P) then
  // determines if the problem is small enough that it can
    // be solved without further splitting
 return Solution(P) // using some method to solve it
else
{
   Divide(P, P1,P2,P3….,Pk)
// divide P into subproblems P1,P2,   Pk k >1
  // Apply DivideandConquer to each of the
// subproblems
   return Combine( DivideAndConquer(P1),
   DivideAndConquer(P2),   ……..
   DivideAndConquer(Pk) )
}
}
```

We get recurrence relation

$$T(n) = \begin{cases} g(n) & \text{if n small} \\ T(n1)+T(n2)+ \ldots T(nk) + D(n) + C(n) & \end{cases}$$

G(n) is the time taken to solve small problem

D(n) is the time taken for dividing

C(n) is the time taken for combining

If everytime problem is divided into k sub problems each of which is 1/k th size of the original problem finally leading to problems of size 1 which need no work, the recurrence relation becomes

$$T(n) = \begin{cases} \theta(1)=\text{constant}=a & \text{if } n=1 \\ b\,T(n/p) + D(n) + C(n) & n > 1 \end{cases}$$

k subproblems all of equal size (n/k) is one possibility but is not always the case

```
Algorithm  BinarySearch(A, m,n, x)
  {  if ( m=n) then      // the small problem is of size 1
         { if( A[m]=x then
                return i
      else  return -1
             }// solution when problem is small
    else
{ mid=(m+n)/2 //dividing the problem into two
  if A[mid]=x  then
   return mid
   else if( A[mid] < x ) then
                return BinarySearch(A, mid+1,n,x)
// solution of subproblem is solution of original problem
     else return BinarySearch(A, m,mid-1,x)
}
}
```

Divides into two subproblems of equal size.
Each subproblem is of size n/2.
There is no effort in dividing . It is done by finding midpoint
There is no effort in combining. Solution of one of the subproblems is the solution of the original problem

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

Merge sort algorithm follows divide and conquer approach
Divide – Divide the input sequence of size n into two subsequences of size n/2 each
Conquer- if sequence is of size 1(small subproblem) there is no work to be done otherwise apply merge sort to the two subsequences
Combine – Merge the sorted subsequences to produce the sorted sequence

```
Algorithm MergeSort( A, m, n)
{ If m < n // indicates there are more than one elements
    { l = (m+n)/2                    //D(n) = θ(1)
          Merge-sort(A, m, l)
           Merge-sort(A, l+1, n)       //    2 T(n/2)
           Merge(A, m, l, n)    //    C(n)=θ(n)
   } //else do nothing -no effort in sorting array of size 1
}
```

Divides into two equal parts

Dividing requires no effort

Combining requires use of merge procedure

```
Algorithm Merge ( A, l, m, n)
{ //one subsequence from l to m and the other from
// m+1 to n
 i=l // pointer at beginning of first subsequence
 j=m+1 // pointer at beginning of second subsequence
k = l  // third pointer for merged sequence
```

```
while  i ≤ m and j ≤ n do
    if A[i] ≤ A[j] then
            B[k] =A[i]   i=i+1
    else  B[k]=A[j]   j=j+1
     k=k+1
while i ≤ m do
 B[k] =A[i]   i=i+1  k=k+1
while j ≤ n  do
 B[k]=A[j]   j=j+1 k=k+1
for i=I to n do A[i]=B[i] // copying B back to A
}
```

At every comparison one element is copied thus the running time of the algorithm is $\theta(n)$

Combining effort is of $\theta(n)$

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

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

$=2(2T(n/4) + bn/2) + bn$

$=4T(n/4) + 2bn$

……..

$=2^kT(n/2k) + kbn$ $\quad n/2^k = 1$

$=2^kT(1) + kbn$

$= na + bnlogn \quad = \theta(nlogn)$

Best case as also worst case of Mergesort takes same time

Mergesort has high space complexity It requires an additional array of same size as also stack space

Merge sort is not an In-place algorithm

Merge sort is stable

Merge sort can be used to sort a linked list of elements- in which case there is no need of additional space but sorting can be done by manipulating pointers

Dividing a linked list into two linked lists is $\theta(n)$

Quick sort algorithm follows divide and conquer approach

Divide – the array A is partitioned into two subarrays A[p,q] and A[q+1, r] such that each element of A[p,q] is less than or equal to each element of A[q+1,r].

Partitioning procedure computes the index q

Conquer- if sequence is of size 1(small subproblem) there is no work to be done otherwise apply quick sort to the two subarrays

Combine – there is no effort required for combining as once the subarrays are sorted, original array also becomes sorted

```
Algorithm QuickSort( A, m,n)
{ if  m < n then
      q= Partition (A,m,n)
      QuickSort( A, m,q)
       QuickSort( A, q+1,n)  }
```

The key to the algorithm is the partition procedure
which rearranges the subarray in place

```
Algorithm Partition( A, m,n)
{ x=A[m]
  i= m-1 // left pointer
  j= n+1 // right pointer
while true do
     repeat j= j-1
        until A[j] ≤ x
      repeat i= i +1
          until A[i] ≥ x
     if  i < j  then
         swap(A[i],A[j])
     else return j
}
```

Partitioning puts elements smaller than pivot element into the bottom region of the array and larger elements into the top region of the array

In partition procedure either left pointer increments or right pointer decrements at every step until the two cross each other

The running time of partition is $\theta(n)$

Performance of QuickSort

Worst case behavior of quicksort occurs when partitioning routine produces one region with 1 element and other with n-1 element and this unbalanced partitioning happens at every step

$$T(n) = T(n-1) + T(1) + \theta(n)$$
$$= T(n-1) + \theta(n)$$
$$= T(n-2) + \theta(n-1) + \theta(n)$$
$$= \sum_{k=1}^{n} \theta(k) = \theta\left(\sum_{k=1}^{n} k\right) = \theta(n^2)$$

Best case behavior of quicksort occurs when partitioning produces two regions of size n/2 and this balanced partitioning happens at every step

$T(n) = 2T(n/2) + \theta(n)$

$= \theta(n\log n)$

The average case running time of quicksort is much closer to the best case than to the worst case.

Worst case is a rare occurrence but surprisingly it occurs when array is already sorted

Average case sorting time of Quick sort is O(nlogn)

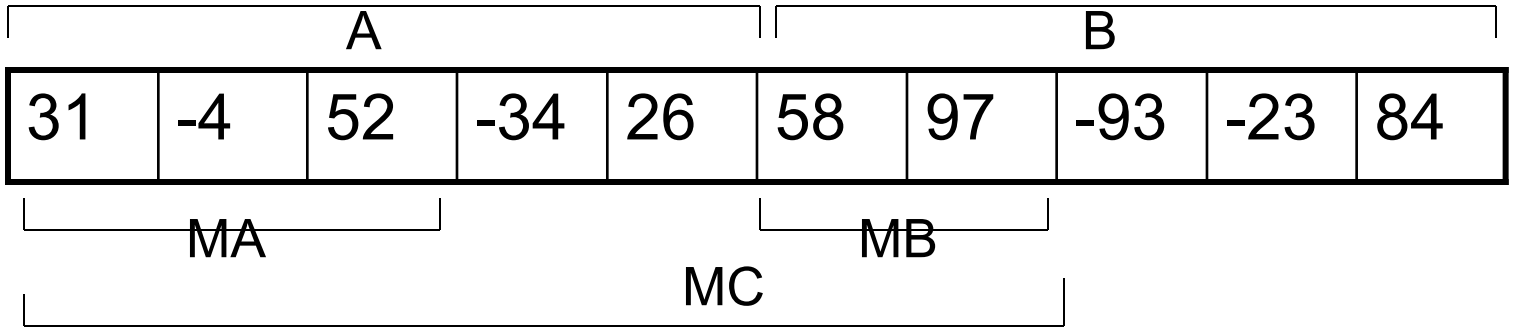
It has high space complexity being recursive

It is an in place algorithm

It is not stable

Quick sort and Merge sort both use divide and conquer strategy

Quick sort spends more time in dividing while merge sort spends more time in combining




```
Algorithm MaxSubvectorDC(A, l, u)
{ if l > u  then return 0.0
If l=u then return max( 0.0, A[l])
   m =( l + u)/2
// to find the maximum at the crossing
        sum = 0.0 maxtoleft=0.0
 for i=m downto l do
  {  sum = sum + A[i]
maxtoleft =max(maxtoleft,sum)}
```

We will consider the running time only in terms of element comparisons

$$T(n) = \begin{cases} 0 & \text{if } n=1 \\ 1 & \text{if } n=2 \\ 2\,T(n/2) + 2 & n > 2 \end{cases}$$

Solving the recurrence relation

$T(n) = 2\,T(n/2) + 2$

$= 2(2T(n/4) + 2) + 2 = 4T(n/4) + 4 + 2$

……..

$= 2^{k-1}T(n/2^{k-1}) + \sum_{1 \le i \le k-1} 2^{i}$

$= 2^{k-1}T(2) + \sum_{0 \le i \le k-1} 2^{i} - 1$ $\qquad n/2^{k-1} = 2$

$= 2^{k-1} + 2^{k} - 2$ $\qquad n = 2^{k}$

$= 3n/2 - 2$

It is the best, worst as average case of comparisons

It is an improvement over the straightforward approach where the number of comparisons were 2n-2

However it has very high space complexity

Strassen's Matrix Multiplication Algorithm

The Matrix Addition algorithm is $\theta(n^2)$

```
Algorithm MatrixAdd( A, B, C, n)
{     for i=1 to n do
       for j=1 to n do
                C[i, j] = A[i, j] +  B[i, j]
               }
```

The Matrix multiplication algorithm is $\theta(n^3)$

```
Algorithm Matrixmult( A, B, C, n)
{     for i=1 to n do
      {  for j=1 to n do
            {   sum=0
                  for k=1 to n do
                     sum= sum+ A[i, k]* B[k, j]
               C[i, j] = sum
            }
     }
}
```

Can Multiplication Algorithm improved by using divide and conquer Approach?

To compute the product C= AB where each A, B and C are n x n matrices ,divide each of the matrices into four matrices of size n/2 x n/2

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{11} = A_{21}B_{12} + A_{22}B_{22}$$

Each of the matrices A_{ij} and B_{ij} should be of same order for matrix multiplication to be feasible. This is possible if n is a power of 2. If it is not, we can pad it with zeroes to make its order equal to nearest power of 2

Divide – Divide each matrix into for matrices of size n/2
Conquer- If matrices are of size1 then matrix multiplication reduces to usual multiplication of two numbers otherwise divide further into submatrices
Combine- The matrix multiplication of the original matrix can then be obtained by performing 8 multiplications of n/2xn/2 matrices and four additions of n/2xn/2 matrices
The recurrence relation for running time is

$$T(n)=\begin{cases} a & n=1 \\ 8T(n/2) + 4\,(n/2)^2 & n>1 \end{cases}$$

$T(n) = 8T(n/2) + bn^2$

$= 8(8\,T(n/4)) + 8\,b(n/2)^2 + b\,n^2$

$= 8^2 T(n/4) + bn^2 (1+ (8/4))$

……

$= 8^k\,T(n/2^k) + bn2\,(1 + 2+\ldots\ldots+2^{k-1})$

$= 8^k\,T(1) + bn^2(2^k -1)$ $\qquad n/2^k = 1$

$=a8^{\log_2 n} + bn^2 (n-1)$

$=an^{\log_2 8} + bn^3 - bn^2$ $\qquad n^{\log a}=a^{\log n}$

$= an^3 + bn^3 - bn^2 = O(n^3)$

Strassen discovered a different approach for combining that requires 7 multiplications of n/2xn/2 matrices and around 18 additions(subtractions) of n2xn/2 matrices This involves computing the seven n/2xn/3 matrices P,Q,R,S,T,U and V as follows

$P= (A_{11}+A_{22})(B_{11}+B_{22})$

$Q= (A_{21}+A_{22})B_{11}$

$R= A_{11}(B_{12} - B_{22})$

$S= A_{22}(B_{21}-B_{11})$

$T=(A_{11}+A_{12})B_{22}$

$U=(A_{21}-A_{11})(B_{11}+B_{12})$

$V=(A_{12}-A_{22})(B_{21}+B_{22})$

$C_{11}=P+S-T+V$

$C_{12}=R+T$

$C_{21}=Q+S$

$C_{22}=P+R-Q+U$

The recurrence relation for running time is

$$T(n)=\begin{cases} a & n=1 \\ 7T(n/2) + 4\,(n/2)^2 & n>1 \end{cases}$$

$$T(n) = 7T(n/2) + bn^2$$

$$= 7(7\,T(n/4)) + 7\,b(n/2)^2 + b\,n^2$$

$$= 7^2 T(n/4) + bn^2 (1 + (7/4))$$

$$\ldots\ldots\ldots\ldots\ldots\ldots$$

$$= 7^k\, T(n/2^k) + bn2\,(1 + (7/4) + \ldots\ldots + (7/4)^{k-1})$$

$$= 7^k\, T(1) + bn^2((7/4)^k - 1) \qquad n/2^k = 1$$

$$= a7^{\log_2 n} + bn^2 ((7/4)^{\log_2 n} - 1) \qquad k = \log_2 n$$

$$= an^{\log_2 7} + bn^2(n^{\log_2 7/4}) - bn^2 \qquad n^{\log a} = a^{\log n}$$

$$= an^{\log_2 7} + bn^2(n^{(\log_2 7 - \log_2 4)}) - bn^2$$

$$= an^{\log_2 7} + b\,n^{2+\log_2 7-2} - bn^2$$

$$= O(n^{\log_2 7}) = O(n^{2.81})$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$
$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{11} = A_{21}B_{12} + A_{22}B_{22}$$

		B11	B21	B12	B22					
	A11	1	0	0	0		0	0	1	0
C 11 =	A12	0	1	0	0	C12 =	0	0	0	1
	A21	0	0	0	0		0	0	0	0
	A22	0	0	0	0		0	0	0	0

		B11	B21	B12	B22					
	A11	0	0	0	0		0	0	0	0
C 21 =	A12	0	0	0	0	C22 =	0	0	0	0
	A21	1	0	0	0		0	0	1	0
	A22	0	1	0	0		0	0	0	1

		B11	B21	B12	B22								
	A11	0	0	0	0		0	0	0	0			
Q =	A12	0	0	0	0	S =	0	0	0	0			
	A21	1	0	0	0		0	0	0	0			
	A22	1	0	0	0		-1	1	0	0			

C21= Q+S

		B11	B21	B12	B22								
	A11	0	0	1	-1		0	0	0	1			
R =	A12	0	0	0	0	T =	0	0	0	1			
	A21	0	0	0	0		0	0	0	0			
	A22	0	0	0	0		0	0	0	0			

C12= R +T

		B11	B21	B12	B22										
	A11	1	0	0	1		0	0	0	0		-1	0	-1	0
P=	A12	0	0	0	0	V =	0	1	0	1	U=	0	0	0	0
	A21	0	0	0	0		0	0	0	0		1	0	1	0
	A22	1	0	0	1		0	-1	0	-1		0	0	0	0

C11=P + S – T + V

C22=P + R –Q + U