DIVIDE AND CONQUER

- We know that some of the problems can be straight-away solved by Brute-Force technique.

- But, in many cases, Brute-force fails. That is, some of the problems can not be solved using Brute force method.

- Here we will study one more algorithm design technique, Divide-and-Conquer.

- The divide and conquer strategy suggests to divide the given problem of size n into k distinct subproblems (1<k<=n).

- These subproblems must be solved and then a method must be found to combine subsolutions to get the solution of whole problem.

- If the subproblem size is relatively large, then, divide and conquer strategy may be re-applied on that.

- Usually, in divide and conquer strategy, we write a control abstraction that mirrors the way an algorithm will look.

- Control Abstraction is a procedure whose flow of control is clear but whose primary operations are specified by other procedures whose precise meanings are left undefined.

- Consider the following algorithm DAndC, which is invoked for a problem P to be solved.

- Small(P) is a Boolean-valued function that determines whether the input size is small enough that the answer can be computed without spitting.
Algorithm DAndC(P)
{
    if Small(P) then
        return S(P);
    else
    {
        divide P into smaller instances P1, P2, …, Pk, k>=1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P1), DAndC(P2), …, DAndC(Pk));
    }
}

• If the size of P is n and the sizes of k subproblems are n1, n2, …, nk, then the computing time of DAndC is described by the recurrence relation:

\[ T(n) = \begin{cases} 
  g(n), & \text{if } n \text{ is small} \\
  T(n1) + T(n2) + \ldots + T(nk) + f(n), & \text{otherwise} 
\end{cases} \]

• The complexity of many divide and conquer algorithms is given by the recurrences of the form –

\[ T(n) = \begin{cases} 
  T(1), & \text{if } n=1 \\
  a.T(n/b) + f(n), & \text{if } n>1 
\end{cases} \]
Merge Sort:
- Merge sort is a best example of DAC technique
- This sorting technique divides a given array \( A[0... n-1] \) by dividing it into two parts, viz. \( A[0... n/2-1] \) and \( A[n/2... n-1] \). Then each of the problems are sorted recursively and finally, they are merged.
- Merging of two sorted arrays is done as follows -
  - Compare first elements of both arrays. Put the smaller element into the resulting array.
  - Now compare remaining element with the 2nd element of other array. Store the smaller into resulting array.
  - Continue the above procedure till one of the arrays get exhausted.
  - Copy all the elements of other array into resulting array.

```
ALGORITHM MergeSort(low, high)
// a[low:high] is an array to be sorted
// Small(P) is true if there is only one element to sort.
{
    if (low<high) then // if there are more than one element
        {
            // Divide P into subproblems
            mid=(low+high)/ 2;
            MergeSort(low, mid); // solve subproblem
            MergeSort(mid+1, high); // solve subproblem
            Merge(low, mid, high) // combine the solutions
        }
}
```
**ALGORITHM Merge(low, mid, high)**

// a[low : high] is an array, and two sorted subsets are a[low : mid] and
// a[mid+1 : high]. The goal is to combine these two sets into a single
// set. b[] is an auxiliary array.

```
{ h:=low, i:=low, j:=mid+1;
  while(h<=mid) and (j<=high) do
  { if(a[h]<=a[j]) then
      b[i] :=a[h];
      h :=h+1;
    else
      b[i] :=a[j];
      j := j+1;
  }
  if  (h>mid) then
    for k := j to high do
      b[i] :=a[k];
      i :=i+1;
  else
    for k := h to mid do
      b[i] :=a[k];
      i :=i+1;
  for k:= low to high do
    a[k] :=b[k];
}
```

**Quick Sort**

- Here, the given array is divided into two sub-arrays such that
  - the elements at the left-side of some key element are less than the key element and
  - the elements at the right-side of the key element are greater than the key element.

- The dividing procedure is done with the help of two index variables and one key element as explained below -

  1. Usually the first element of the array is treated as key. The position of the second element is taken as the first index variable left and the position of the last element will be the index variable right.

  2. Now the index variable left is incremented by one till the value stored at the position left is greater than the key.

  3. Similarly right is decremented by one till the value stored at the position right is smaller than the key.
4. Now, these two elements are interchanged. Again from the current position, left and right are incremented and decremented respectively and exchanges are made appropriately, if required.

5. This process is continued till the index variables either meet or crossover. Now, exchange key with the element at the position right.

6. Now, the whole array is divided into two parts such that one part is containing the elements less than the key element and the other part is containing the elements greater than the key. And, the position of key is fixed now.

7. The above procedure (from step i to step vi) is applied on both the sub-arrays. After some iteration we will end-up with sub-arrays containing single element. By that time, the array will be stored.

ALGORITHM QuickSort (p, q)
// Sorts the elements a[p], ..., a[q] which is inside the array a[1: n].
{
  if(p<q) then
  {
    j :=Partition(a, p, q+1);
    QuickSort(p, j-1);
    QuickSort(j+1, q);
  }
}
ALGORITHM Partition(a, m, p)
{
    v := a[m];
    i := m;
    j := p;
    repeat
        repeat
            i := i + 1;
            until (a[i] >= v);
        repeat
            j := j - 1;
            until (a[j] <= v);
        if (i < j) then
            Interchange(a, i, j);
    } until (i >= j);
    a[m] := a[j];
a[j] := v;
return j;
}

Algorithm Interchange(a, i, j)
{
    p := a[i];
a[i] := a[j];
a[j] := p;
}

BINARY SEARCH

ALGORITHM BinSrch(a, i, l, x)
// Given an array a[i:l] of elements in nondecreasing order,
// 1 <= i <= l, determine whether x is present. If so, return
// such that x = a[j], else return 0.
{
    if (l == i) then // If Small(P)
        {    if (x == a[i]) then return i;
            else return 0;
        }
    else
        {    mid := (i + l) / 2;
            if (x == a[mid]) then return mid;
            else if (x < a[mid]) then
                return BinSrch(a, i, mid - 1, x);
            else return BinSrch(a, mid + 1, l, x);
        }
}

Defective Chess Board Problem

- A chessboard is an $n \times n$ grid, where $n = 2^k$

- A defective chessboard is a chessboard that has one unavailable (defective) position.
• The problem is to tile (cover) all non-defective cells using a triomino.

• A triomino is an L shaped object that can cover three squares of a chessboard.

• A triomino has four orientations:

Tiling A Defective Chessboard

Place \((n^2 - 1)/3\) triominoes on an \(n \times n\) defective chessboard so that all \(n^2 - 1\) nondefective positions are covered.
Divide into four smaller chessboards. $4 \times 4$

One of these is a defective $4 \times 4$ chessboard.

Make the other three $4 \times 4$ chessboards defective by placing a triomino at their common corner. Recursively tile the four defective $4 \times 4$ chessboards.
In general, a $2^k \times 2^k$ defective chessboard can be divided as –

\[
\begin{array}{|c|c|}
\hline
2^{k-1} \times 2^{k-1} & 2^{k-1} \times 2^{k-1} \\
\hline
2^{k-1} \times 2^{k-1} & 2^{k-1} \times 2^{k-1} \\
\hline
\end{array}
\]

Analysis

• Let $n = 2^k$.
• Let $t(k)$ be the time taken to tile a $2^k \times 2^k$ defective chessboard. Then,

\[
t(k) = \begin{cases} 
0, & \text{if } n = 0 \\
4t(k-1) + c, & \text{otherwise}
\end{cases}
\]

• Here, $c$ is constant representing time spent on finding the appropriate position for a triomino and to rotate the triomino for a required shape.
\[ t(k) = 4t(k-1) + c \]
\[ = 4[4t(k-2) + c] + c \]
\[ = 4^2 t(k-2) + 4c + c \]
\[ = 4^2[4t(k-3) + c] + 4c + c \]
\[ = 4^3 t(k-3) + 4^2c + 4c + c \]
\[ = \ldots \]
\[ = 4^k t(0) + 4^{k-1}c + 4^{k-2}c + \ldots + 4^2c + 4c + c \]
\[ = 4^{k-1}c + 4^{k-2}c + \ldots + 4^2c + 4c + c \]
\[ = \Theta(4^k) \]
\[ = \Theta(\text{number of triominoes placed}) \]