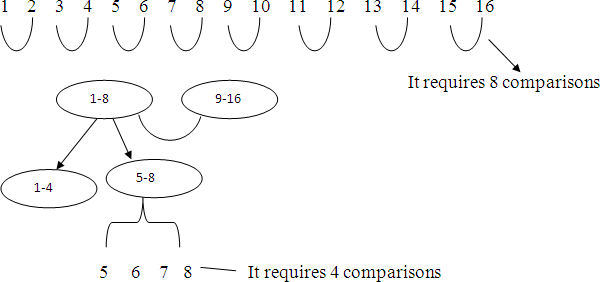
**Unit-2**

**DIVIDE AND CONQUER METHOD**

It is one of the methods for generating or developing algorithms to solve problems. The following procedure is used to get solution or to develop an algorithm for any problem. If p is the problem to be solved then check whether p can be directly solvable. If the problem p can be solved directly then solve the problem and get the solution. Otherwise, divide the problem p into number of sub problems (p1, p2, p3,…). If the sub problems can be directly solvable then solve the sub problems and combine the solutions to get solution for entire problem. Otherwise, continue the division process until the directly solvable sub problems are obtained. Combining the solutions of sub problems is necessary for only some of the problems.

## Ex:- A bag contains 16 coins. One coin is fake coin. There is a weighing machine. The fake coin has to be identified using weighing machine. 8 comparisons are required to identify the fake coin.

Using divide and conquer method, 4 comparisons are enough to identify the fake coin.



# Image result for divide and conquer algorithm example

# Control Abstraction of Divide and Conquer Method: Control abstraction of Divide and Conquer method indicates the general procedure used to solve any problem or to develop an algorithm for any problem using Divide and Conquer method. Control abstraction of divide and conquer method is

Algorithm D&C(p)

// p is the problem to be solved

{

if small(p) then solve(p) else

{

Divide p into number of sub problems p1, p2,…..pn D&C(p1);

D&C(p2);

.

D&C(pn);

Combine solutions of sub problems

}

}

**Small(**) is a user defined function which returns true if the problem is small enough to solve directly, otherwise it returns false.

**Solve()** is a user defined function which gets solution of the problem.

# Applications of Divide and Conquer

Following are the important applications of divide and conquer method

1. Binary Search
2. Quick Sort
3. Merge Sort

# 🡪 Binary Search: It is one of the techniques used to check the presence of a value in a set or list of values. The values in the list or set should be in sorted order. The procedure used in this method is as follows: the value to be searched is compared with the middle value in the list. If they match then the process is terminated and announced that the value is identified. Otherwise, the list is divided into two equal size parts based on middle position and then the value is searched in first part if the value to be searched is less than the middle value of list or in second part if the value to be searched is greater than the middle value of list. This process is repeated until the value is found or the list shrinks to a single value.

## Algorithm

Algorithms Binsearch ( a, n, key, low, high)

{

// a is array of size n,Key is the element to be searched

// if key is found then return j, such that key = a[i] otherwise return -1

if ( low < high) then

{

mid: = (low + high)/2;

if ( key = a[mid]) then Return mid;

else if (key < a[mid])

Binsearch ( a, n, key, low, mid-1);

else

if ( key > a[mid])

Binsearch ( a, n, key, mid+1, high);

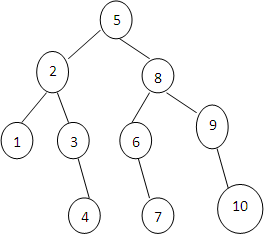
}

return -1;

}

## Time complexity: To calculate time complexity, a tree called decision tree is constructed. A decision tree is constructed based on mid position of list as well as sub lists that will be generated in the searching process.

**Ex**:- If the number of elements in the list is n=10 then the decision tree is



The decision tree indicates the number of comparisons required to identify values at different positions of the list. For example, one comparison is required to identify the value at 5th position of the list. Two comparisons are required to identify the value at 2nd position. Three comparisons are required to identify the value at 6th position.

Time complexity of binary search is calculated by considering 2 cases

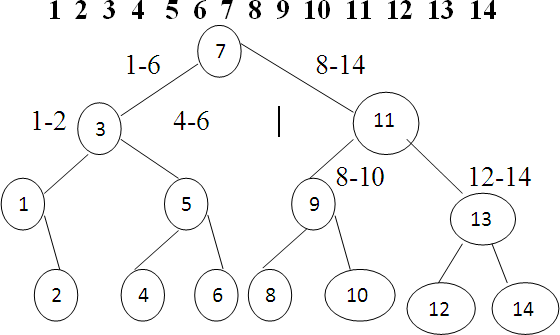
**Case 1:** Successful Search (value is present in the list) We consider 3 cases here

1. **Best Case**: Best case is encountered when the value is present at middle position of the list. One comparison is enough to identify the value at middle position. So, time complexity is **O(1).**
2. **Worst Case**: Worst case is encountered when the value is present at any position of the list. To identify the value at any position, a maximum of four (height of the decision tree+1) comparisons are required. Height of decision tree with n nodes is log2n. So, time complexity is **O(log2n).**
3. **Average Case:** Average case is encountered when the value is present at any position of the list. To identify the value at any position, a maximum of four (height of the decision tree+1) comparisons are required. Height of decision tree with n nodes is log2n. So, time complexity is **O(log2n).**

**Case 2:** Unsuccessful Search (value is not present in the list)

To know that the value is not present in the list four (height of the decision tree+1) comparisons are required. Height of decision tree with n nodes is log2n. So, time complexity is **O(log2n).**

**Ex:** Construct decision tree for 14 positions.



# 🡪 Merge Sort

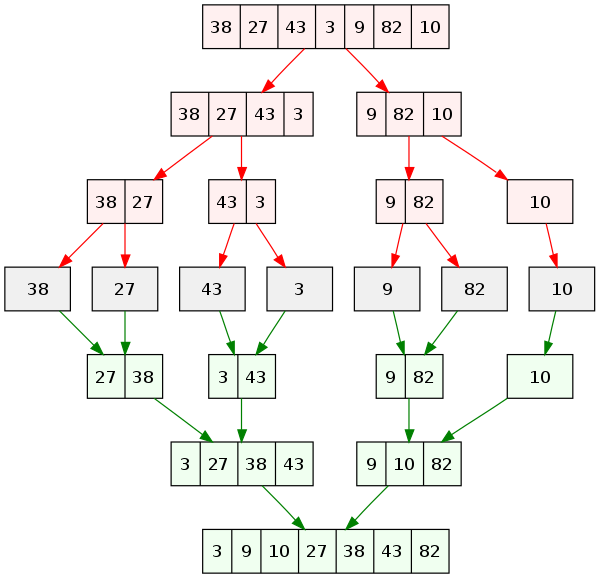
Merge sort is one of the techniques used to sort a list of values or to arrange a list of values in ascending order. The process for sorting a list of values is as follows:

Divide the list into two equal parts based on mid position. Divide each part into two equal parts again. Continue this division process until each part contains only one value. Now, combine the parts in reverse direction. While combining two parts, compare the values in two parts and place them in sorting order.

To combine or merge two parts, use the following procedure:

Set a pointer (i) at the beginning of first part. Set a pointer (j) at the beginning of second part. Compare the value at ith position with the value at jth position. If the value at ith position is less than or equal to the value at jth position then place the value at ith position into the temporary array and then move the ith pointer to the next position. Otherwise, place the value at jth position into the temporary array and then move the jth pointer to the next position. Repeat this procedure until one of the parts is completed. When one of the parts (first or second) is completed then place the values in other part one by one into the temporary array.

Finally, copy the values in temporary array into the original array. Ex: sort the following list of values using merge sort technique



## Algorithm of MergeSort

Algorithm MergeSort(a, s, e)

// a is an array containing list of values to be sorted

// s is starting position and h is ending position of the list

{

if s < e then

{

m := (s+e)/2; MergeSort(a, s, m); MergeSort(a, m+l, e); Merge(a, s, m, e);

}

}

Algorithm Merge(a, s, m, e)

// s is starting position of first part, m is ending position of first part

// e is ending position of second part

{

// b is an array used to store values

i := s;

j := m+1;

k := s;

while i ≤ m and j ≤ e do

{

if a[i] ≤ a[j] then

{

}

else

{

}

}

b[k] := a[i]; i := i+1;

k := k+1;

b[k] := a[j]; j := j+1;

k := k+1;

while i ≤ m do

{

b[k] := a[x]; i := i+1;

k := k+1;

}

while j ≤ e do

{

b[k] := a[x]; j := j+1;

k := k+1;

}

for x := s to e do

{

a[x] := b[x];

}

}

## Time Complexity of Merge Sort Algorithm

Merge sort algorithm sorts a list of values by dividing the list into two parts, sorting the two parts separately and combining the two parts.

Time for sorting the list with n values=time for sorting the first part with n/2 values+ time for sorting the second part with n/2 values+ time for merging the two parts T(n)=T(n/2)+T(n/2)+time of merge algorithm

Time complexity of Merge algorithm is 3+5\*n/2+4\*n/2+2\*n For declaration statements, count is incremented by 3

For „while‟ statement, count is incremented by 5\*n/2

For „for‟ statement (first loop or second loop), count is incremented by 4\*n/2 For „for‟ statement (last for loop), count is incremented by 2\*n

3+5\*n/2+4\*n/2+2\*n=3+9n/2+2n=cn=O(n) Where c is some constant.

Time complexity of Merge Sort algorithm is T(n)=T(n/2)+T(n/2)+cn

T(n)=2T(n/2)+cn T(n)=2[2T(n/2)+cn/2]+cn T(n)=22 T(n/22)+cn+cn T(n)=22 T(n/22)+2cn

=22 T(2T(n/8)+cn/4]+2cn

=23T(n/23)+cn+2cn

=23T(n/23)+3cn

:

:

After k times

=2kT(n/2k)+kcn

Assuming that n=2k then k= log2n T(n)=nT(n/n)+cn(log2n)=nT(1)+cnlog2n=n+cnlog2n=**O(nlog2n)**

# 🡪 Quick Sort Quick sort is a technique to sort a list of values. The procedure for sorting a list of values is as follows:

Select a value (generally the first value) in the list as pivot value. Set a pointer (i) to the starting position of list. Set a pointer (j) to the ending position of list. Move the pointer „i‟ in forward direction until the value at ith position is less than or equal to the pivot value.

Move the pointer „j‟ in back word direction until the value at jth position is greater than the pivot value. If the position of „i‟ is less than the position of „j‟ then swap the values at „i‟ and „j‟ positions and then move the

„i‟ and „j‟ pointers again. Otherwise, swap the pivot value with the value at jth position and divide the list into two parts based on position „j‟. The first part includes the values from the starting of list to (j-1)th position and the second part includes the values from (j+1)th position to ending of list. Now, sort the first and second parts separately using same procedure.

Ex: sort the following list of values using quick sort technique

# Algorithm

Algorithm Quicksort(a, s, e)

// a is an array containing list of values to be sorted

// s is starting position and e is ending position of list

{

if s < e then

{

j := Partition(a, s, e); Quicksort(a, s, j-1); Quicksort(a, j+1, e);

}

}

Algorithm Partition(a, s, e)

{

p := a[s]; i := s;

j := e;

while i < j do

{

while a[i] ≤ p do

i := i+1;

while a[j] > p do

j := j-1;

if i < j then

{

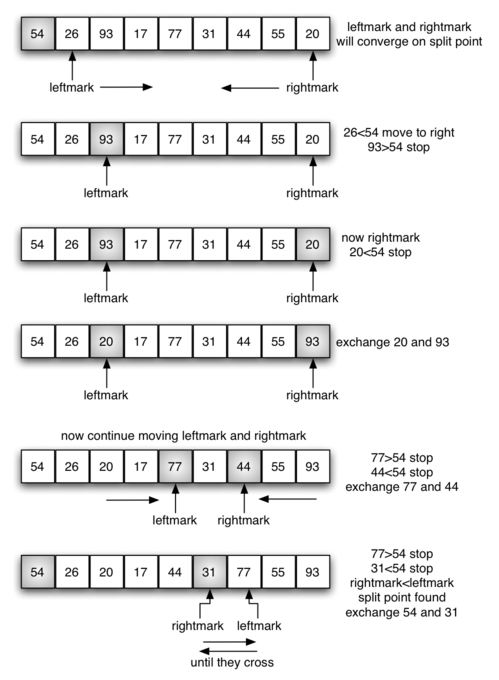
t := a[i]; a[i] := a[j]; a[j]: = t;

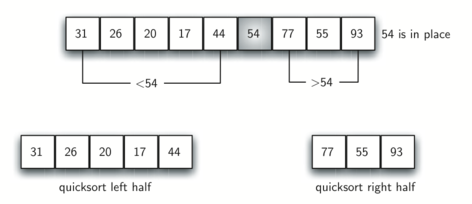
}

}

t := a[s]; a[s] := a[j]; a[j] := t; return j;

}





## Time Complexity

**Time Complexity of Partition Algorithm:**

Partition algorithm moves the pointer „i‟ towards right for a maximum of „n‟ positions and the pointer „j‟ towards left for a maximum of „n‟ positions. So, time complexity of partition algorithm is n+n+c=2n+c=O(n).

Where „c‟ is some constant.

## Time Complexity for Quick Sort Algorithm:

If the list is containing „n‟ number of values then the time complexity for sorting the „n‟ values is T(n)=Time for partition + Time for sorting first part + Time for sorting second part T(n)=cn+T(i)+T(n-1-i) Assuming that the number of values in the first part is „i‟.

Time complexity is analyzed in 3 cases

## Worst Case

Worst case is encountered when we try to sort the values which are already in sorting order. Ex: 10 20 30 40 50 60

p i j

10 20 30 40 50 60

p j i

When the list is divided into two parts then the first part includes no values, second part includes (n-1) values.

T(n)=cn+T(0)+T(n-1-0) T(n)=cn+0+T(n-1)

T(n)=T(n-1)+cn T(n)=T(n-2) +c(n-1) +cn

T(n)=T(n-2)+2cn-c

T(n)=T(n-3)+3cn-3c

.

.

After n times T(n)=T(n-n)+ncn-nc T(n)=0+cn2-nc

T(n) =cn2-cn T(n)=**O(n2)**

## Best Case

Best case is encountered when the list is divided into 2 equal size parts as in case of merge sort algorithm. So, Time complexity is

T(n)=cn+T(n/2)+T(n/2) T(n)=2T(n/2)+cn T(n)=2[2T(n/2)+cn/2]+cn T(n)=22 T(n/22)+cn+cn T(n)=22 T(n/22)+2cn

=22 T(2T(n/8)+cn/4]+2cn

=23T(n/23)+cn+2cn

=23T(n/23)+3cn

:

:

After k times

=2kT(n/2k)+kcn

Assuming that n=2k then k= log2n T(n)=nT(n/n)+cn(log2n)

=nT(1)+cnlog2n

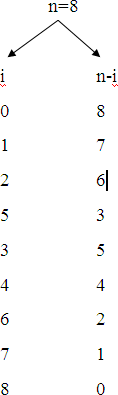
=n+cnlog2n

=**O(nlog2n)**

## Average Case

Average case is encountered when the list is divided into two unequal size parts. The average case time complexity is calculated by considering all possible sizes for first part & second part.

Ex: if the list is containing 8 values the possible sizes of first and second parts are



Time complexity is

T(n)=cn+[T(0)+T(1)+……….+T(n)]/n+[T(n)+T(n-1)+………+T(0)]/n T(n)=cn+2/n[T(0)+T(1)+ +T(n)]

After solving the above equation, the time complexity is T(n)=**O(nlog2n)**

## 🡪 Randomized Quick Sort: Randomized quick sort is a variant of quick sort. In Quick sort, generally the first value of the list is selected as pivot element. In randomized quick sort, any value of the list can be selected as pivot element. While sorting different parts of list, values at different positions of the lists can be selected as pivot elements. For example, for sorting first part of the list the value at third position can be selected, for sorting second part of the list the value at second position can be selected.

## Algorithm

Algorithm RandomQuickSort(a, s, e)

// a is an array containing the list of values to be sorted

// s is starting position of array and e is ending position of array

{

if s < e then

{

j := Partition(a, s, e); RandomQuickSort(a, s, j-1); RandomQuickSort(a, j+1, e);

}

}

Algorithm Partition(a, s, e)

{

k := Random(s, e);

p := a[k]; i := s;

j := e;

while i < j do

{

while a[i] ≤ p do

i := i+1;

while a[j] > p do

j := j-1;

if i < j then

{

t := a[i]; a[i] := a[j]; a[j]: = t;

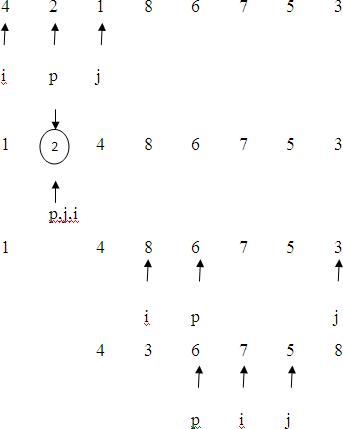
}

}

t := a[s]; a[s] := a[j]; a[j] := t; return j;

}

Random(s, e) is a user defined function which returns a value between s and e.



**🡪 Min Max Problem:** A divide-and-conquer algorithm for this problem would proceed as follows:

Let P = (n,a[i],….,a[j]) denote an arbitrary instance of the problem. Here n is the number of elements in the list a[i],….,a[j] and we are interested in finding the maximum and minimum of this list.

Let small(P) be true when n ≤ 2.

If n = 1, the maximum and minimum are a[i].

If n = 2, the problem can be solved by making one comparison.

If the list has more than two elements, P has to be divided into smaller instances.

For example, we might divide P into the two instances

P1 = (n/2,a[1],….,a[n/2]) and P2 = (n - n/2,a[n/2 + 1],….,a[n]).

After having divided P into two smaller sub problems, we can solve them by recursively invoking the same divide and conquer algorithm.

We can combine the Solutions for P1 and P2 to obtain the solution for P as follows.

If MAX(P) and MIN(P) are the maximum and minimum of the elements of P, then MAX(P) is the larger of MAX(P1) and MAX(P2) also MIN(P) is the smaller of MIN(P1) and MIN(P2).

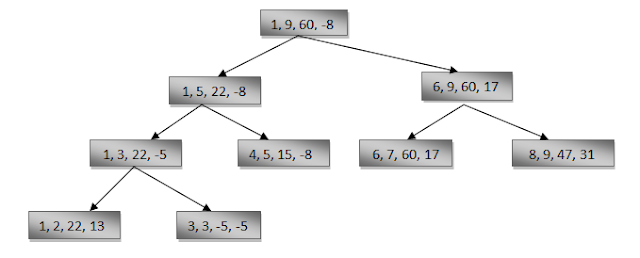
MaxMin is a recursive algorithm that finds the maximum and minimum of the set of elements {a(i),a(i+1),…,a(j)}. The situation of set sizes one (i=j) and two (i=j-1) are handled separately. For sets containing more than two elements, the midpoint is determined and two new sub problems are generated. When the maxima and minima of these sub problems are determined, the two maxima are compared and the two minima are compared to achieve the solution for the entire set.

 Suppose we simulate MaxMin on the following nine elements:

Ex: a: [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]

    22   13   -5   -8   15  60   17   31  47

A good way of keeping track of recursive calls is to build a tree by adding a node each time a new call is made. On the array a[ ] above, the following tree is produced.

[](http://3.bp.blogspot.com/-BntLQuCaTaM/UDtYks9NG0I/AAAAAAAAAI4/c3L99uVf2oE/s1600/Capture.PNG)

We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

Alogrithm MaxMin(i, j, max, min)

// a[1:n] is a global array. Parameters i and j are integers,

// 1≤i≤j≤n. The effect is to set max and min to the largest and // smallest values in a[i:j].

{

if (i=j) then max := min := a[i]; //Small(P)

else if (i=j-1) then // Another case of Small(P)

{

if (a[i] < a[j]) then max := a[j]; min := a[i];

else max := a[i]; min := a[j];

}

else

{

// if P is not small, divide P into sub-problems.

// Find where to split the set.

mid := ( i + j )/2;

// Solve the sub-problems.

MaxMin( i, mid, max, min );

MaxMin( mid+1, j, max1, min1 );

// Combine the solutions.

if (max < max1) then max := max1;

if (min > min1) then min := min1;

}

}

Complexity: Now what is the number of element comparisons needed for MaxMin? If T(n) represents this number, then the resulting recurrence relation is

0 n=1

T(n) = 1 n=2

T(n/2) + T(n/2) + 2 n>2

When n is a power of two, n = 2k -for some positive integer k, then

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

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

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

.

.

= 2k-1 T(2) + ∑(1≤i≤k-1) 2k

= 2k-1 + 2k – 2

= 3n/2 – 2 = O(n)

Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

🡪 **Defective Chess Board:** A chessboard is an n x n grid, where n is a power of 2. A defective chessboard is chessboard that has one unavailable (defective) position.

Given a n by n board where n is of form 2k where k >= 0. The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

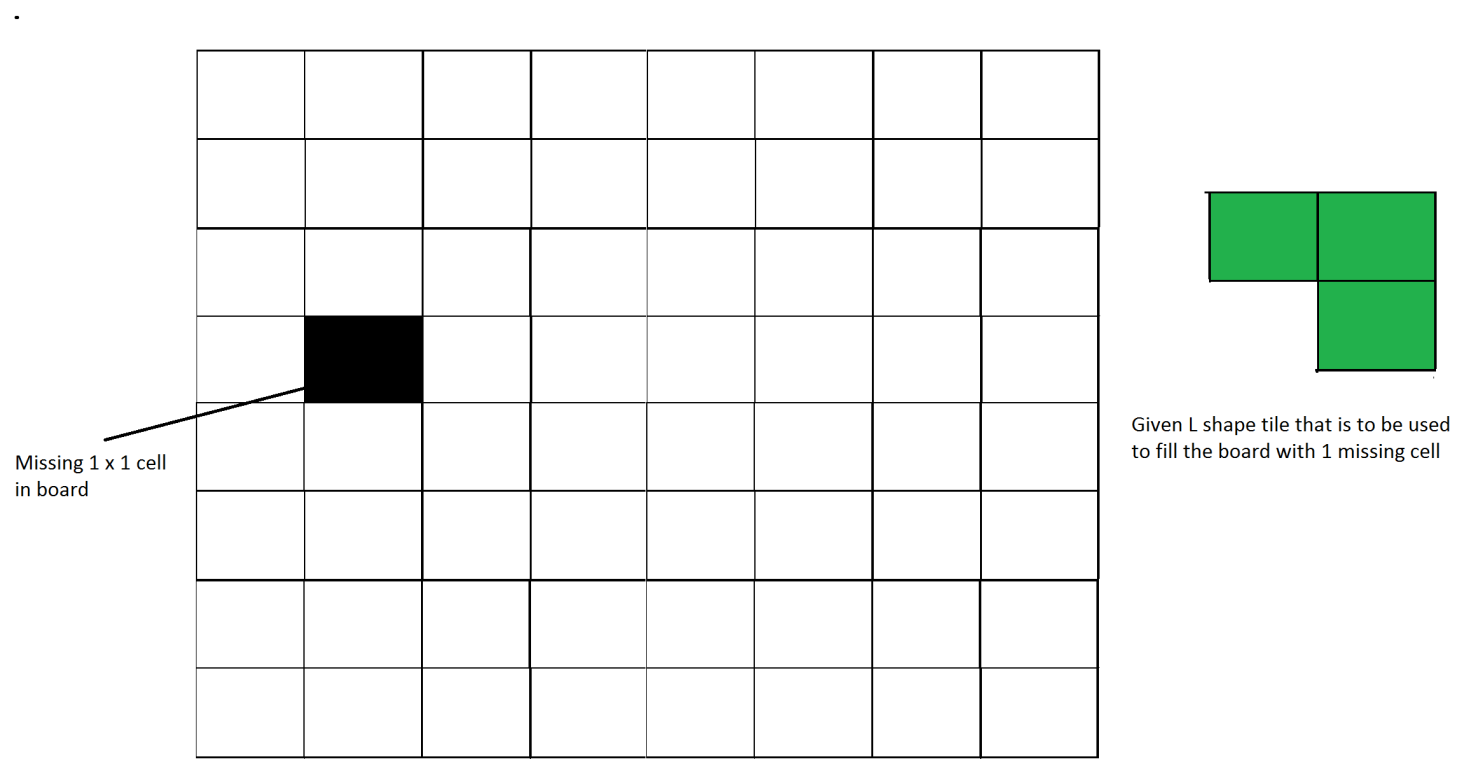


Figure 1: An example input

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



Have to place (n2-1)/3 triominoes on an n x n defective chessboard so that all n2-1 non-defective positions are covered.

This problem can be solved using Divide and Conquer. Below is the recursive algorithm.

// n is size of given square, p is location of missing cell

Tile(int n, Point p)

1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing but a tile and can be filled with a single tile.

2) Place a L shaped tile at the center such that it does not cover the n/2 \* n/2 subsquare that has a missing square. Now all four subsquares of size n/2 x n/2 have a missing cell (a cell that doesn't need to be filled). See figure 2 below.

3) Solve the problem recursively for following four. Let p1, p2, p3 and p4 be positions of the 4 missing cells in 4 squares.

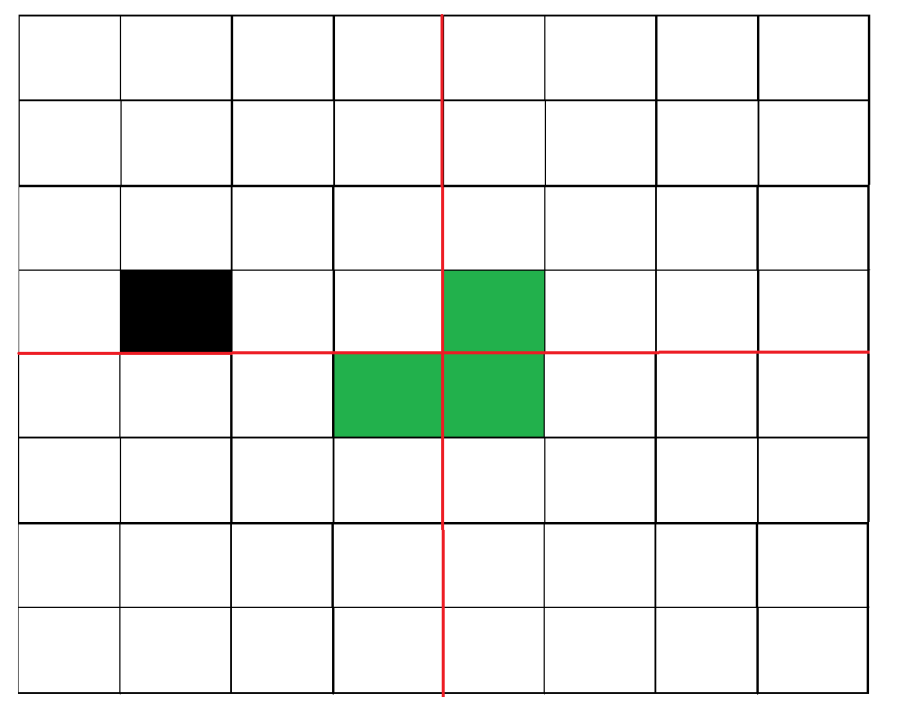
a) Tile(n/2, p1)

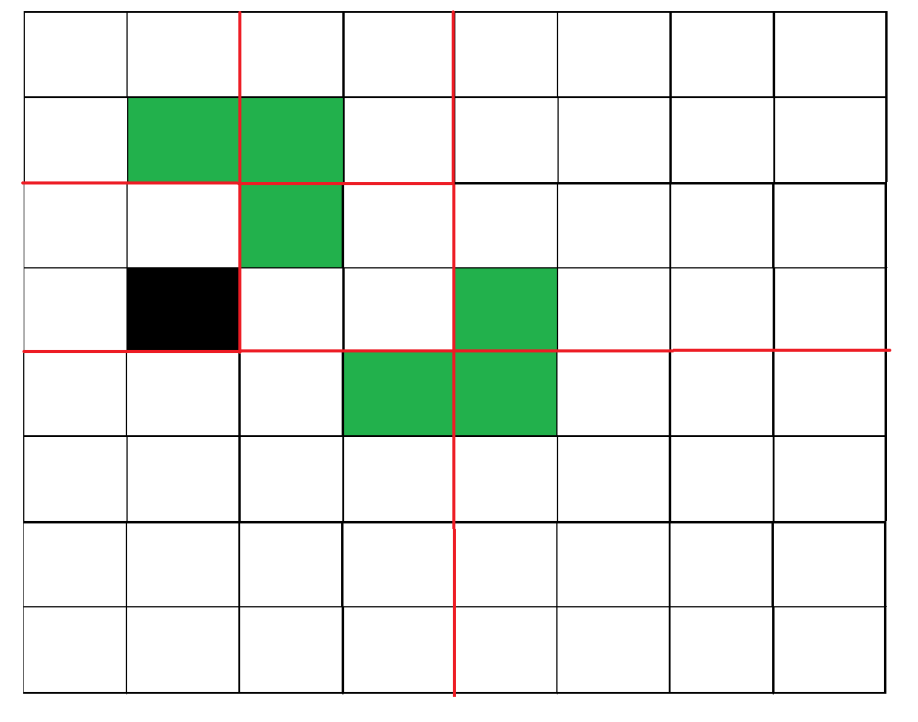
b) Tile(n/2, p2)

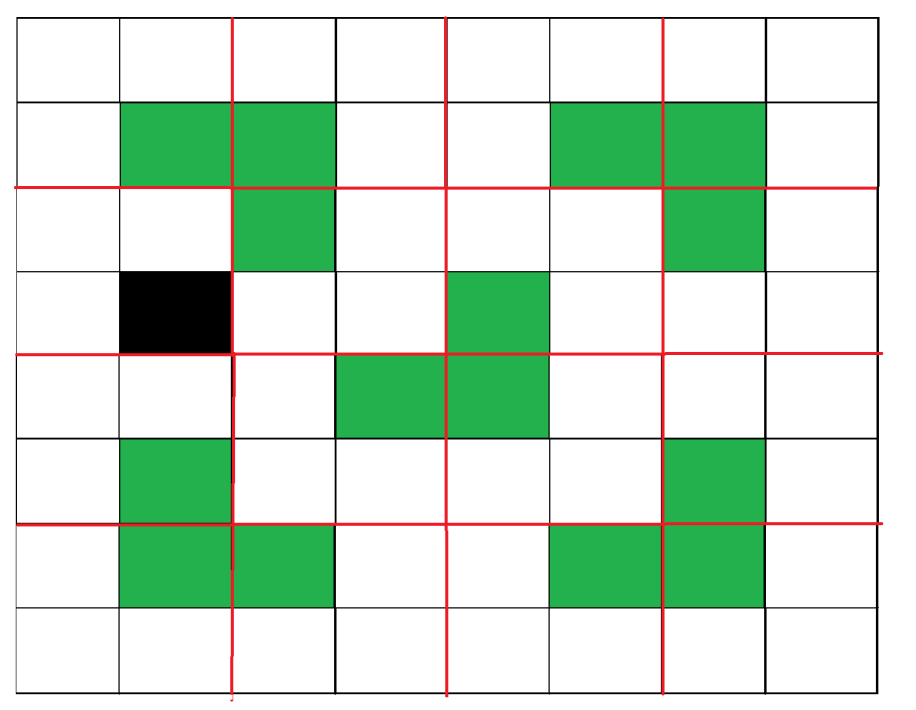
c) Tile(n/2, p3)

d) Tile(n/2, p3)

The below diagrams show working of above algorithm

[](https://www.geeksforgeeks.org/wp-content/uploads/tiles3.png)  
Figure 2: After placing first tile

[](https://www.geeksforgeeks.org/divide-and-conquer-set-6-tiling-problem/tiles4/)  
Figure 3: Recurring for first subsquare.

[](https://www.geeksforgeeks.org/divide-and-conquer-set-6-tiling-problem/tiles5/)  
Figure 4: Shows first step in all four sub squares.

**Time Complexity:**

Recurrence relation for above recursive algorithm can be written as below. C is a constant.

T(n) = 4T(n/2) + C n>=2.

T(2) = 1