**Algorithms-Binary Search**

**Objective :** To find an element in an sorted array

**Input:** A sorted array, arrA[] and an key

**Output :** Return true if element is found, else false.

**Approach:** The idea is to compare the middle element of array with the key, if key equal to the middle element , that’s it you have find your element, return true. If key is greater than the middle element, chuck out the first half of the array, you wont find your key in the first half and do the recursive search on the right half of the array and vice versa.

If(mid\_element==key)

return true;

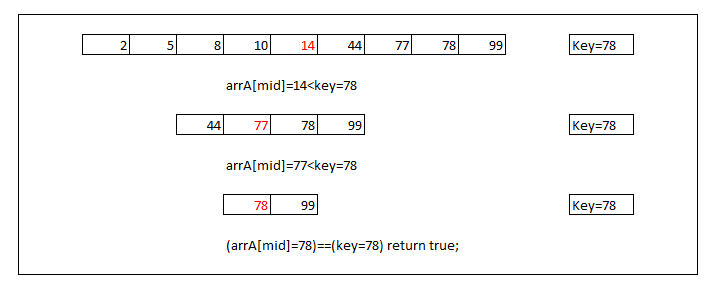
else if (mid<key)

do recursive search on the right half of the array.

else

do recursive search on the left half of the array.

**Time Complexity**: O(logN) –since we are eliminating half of the array with every comparison.



**Complete Code:**

**package** interviewQuestion;

**public** **class** BinarySearch {

**private** **int** [] arrA;

**private** **int** number;

**public** BinarySearch(**int** [] arrA){

**this**.arrA = arrA;

}

**public** Boolean Search(**int** low,**int** high, **int** number){

**if**(low>high){

**return** **false**;

}

**int** mid = (high+low)/2;

**if**(arrA[mid]==number)**return** **true**;

**else** **if** (arrA[mid]>number) **return** Search(low,mid-1,number);

**else** **return** Search(mid+1,high,number);

}

**public** **static** **void** main(String args[]){

**int** [] a = {2,5,8,10,14,44,77,78,99};

**int** number = 99;

BinarySearch b = **new** BinarySearch(a);

System.*out*.println("The "+ number + " present in array a ??? :" + b.Search(0, a.length-1, number));

number = 76;

System.*out*.println("The "+ number + " present in array a ??? :" + b.Search(0, a.length-1, number));

}

}

Output:

The 99 present in array a ??? :true

The 76 present in array a ??? :false

Download link:

**Algorithms – Merge Sort**

**Objective :** To sort elements in an array

**Input:** A insorted array, arrA[].

**Output :** A sorted array.

**Approach:**

**Divide and Conquer:** In this approach we divide the main problems into smaller problems, solve them and merge the results to get the final result.

**How Divide and conquer works in Merge Sort:**

We divide the elements into two half’s by middle of the array. We solve the left half and right half recursively and merge the results.

**Merging:**

Once the sorting is done individually on both the half’s, our next task will be merge them. To merge we start with both the arrays at the beginning, pick the smaller one put into array and then compare the next elements and so on.

4

9

10

5

3

6

1

2

6

5

4

3

2

1

10

9

5

4

6

3

2

1

10

9

9

4

6

3

2

1

10

5

4

9

10

5

3

6

1

2

4

9

10

5

3

6

1

2

4

9

10

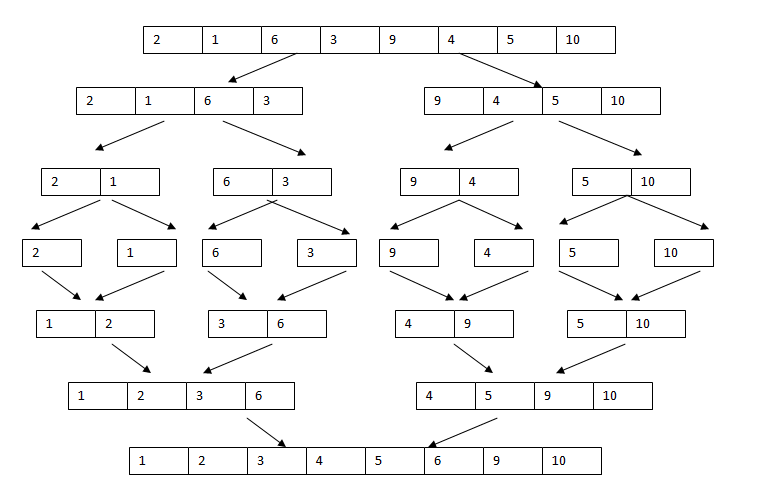
5

3

2

6

1



**Time Complexity : O(nlogn) { O(logn) for dividing and O(n) for merging.**

**Note: we can make merging more efficient by implementing these approaches**

**Using Auxiliary Array with copying data** – In this approach you wont create new array everytime for merging instead you create Auxiliary array. This will save memory for you.

**Alternate Merging Between Primary and Auxiliary Array:** This is the best approach for merging. You don’t copy the entire array to auxiliary array for merging instead you do alternate merging between main array and auxiliary array.

Below is the running time comparison between all three approaches

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Size** | **Dynamic Memo Allocation Algo** | **Using Auxillary Array with copying data** | **Alternate Merging Between Primary and Auxillary Array** |
| 1 Million | 600-630 mili sec | 450-470 mili sec | 400-425 mili sec |
| 10 million | 6 secs | 4.2 secs | 2.3 secs |
| 100 million | 56 secs | 46 sec | 18 sec |

**You can find the implementation of all these approaches here –**

<https://github.com/SumitJainUTD/DataStructuresAlgos/tree/master/3%20Different%20Impl%20of%20Merge%20Sort>

**Complete Code:**

**package** interviewQuestion;

**public** **class** MergeSort {

**private** **int** arrSize;

**private** **int** [] arrAux;

**private** **int** [] arrInput;

**public** MergeSort(**int** [] arrInput){

**this**.arrInput = arrInput;

arrSize = arrInput.length;

arrAux = **new** **int** [arrSize];

}

**public** **int**[] mergeSorting(){

sort(0,arrSize-1);

**return** arrInput;

}

**public** **void** sort(**int** low, **int** high){

**if**(low<high){

**int** mid = low+((high-low))/2;

sort(low,mid);

sort(mid+1,high);

merge(low, mid, high);

}

}

**public** **void** merge(**int** low, **int** mid, **int** high){

//copy the entire array in the Auxilary array

**for**(**int** i=low;i<=high;i++){

arrAux[i] = arrInput[i];

}

**int** i = low;

**int** j = mid+1;

**int** k = low;

**while**(i<=mid && j<=high){

**if**(arrAux[i]<=arrAux[j]){

arrInput[k]=arrAux[i];

i++;

}

**else**{

arrInput[k]=arrAux[j];

j++;

}

k++;

}

**while**(i<=mid){

arrInput[k]=arrAux[i];

i++;

k++;

}

}

**public** **void** displayArray(**int** [] b){

**for**(**int** i=0;i<b.length;i++){

System.*out*.print(" " + b[i]);

}

}

**public** **static** **void** main(String[] args){

**int** [] a = {2,1,6,3,9,4,5,10};

MergeSort m = **new** MergeSort(a);

**int** [] b = m.mergeSorting();

m.displayArray(b);

}

}

Output :

1 2 3 4 5 6 9 10

**Algorithms - Hash Table Implementation**

**Objective :** To implement a Hash Table

**Input:** A set of pairs of keys and values

**Approach:**

* **Create a Hash Table**
  + Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
* **Insert values in hash table using put(key,value)**
  + ht.put(key, value);
* **Get values from hash table using get(key)** 
  + ht.get(key);

|  |  |
| --- | --- |
| Key | Value |
| 1 | Sumit |
| 2 | Raghav |
| 3 | Rishi |

hashTable Object  
Advantage : The search time for any element is O(1) since it uses key to find an element so it takes constant time. But drawback is that it takes extra space.

**Complete Code:**

**package** interviewQuestion;

**import** java.util.Hashtable;

**public** **class** SimpleHashTable {

**int** [] a = **new** **int**[5];

String [] arrNames = **new** String[]{"Sumit","Jain","Raghav","Garg","Gaurav","Rishi"};

Hashtable<Integer, String> ht = **new** Hashtable<Integer, String>();

**public** **void** insertValues(){

**for**(**int** i=0;i<arrNames.length;i++ ){

ht.put(i+1,arrNames[i]);

}

}

**public** String getValue(**int** key){

**return** ht.get(key);

}

**public** **static** **void** main (String [] args){

SimpleHashTable sht = **new** SimpleHashTable();

sht.insertValues();

System.*out*.println("All values inserted");

System.*out*.println("Employee with ID 1 is "+ sht.getValue(1));

System.*out*.println("Employee with ID 3 is "+ sht.getValue(6));

}

}

Output:

All values inserted

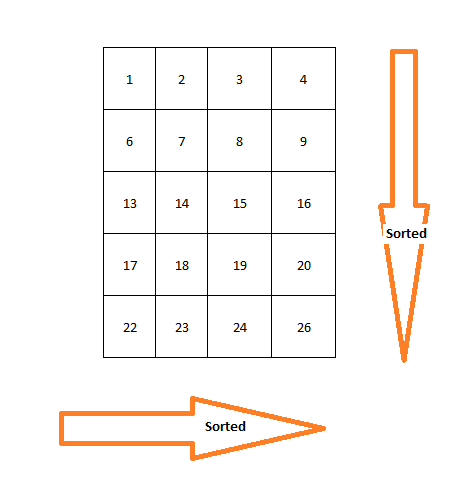
Employee with ID 1 is Sumit

Employee with ID 3 is Rishi

**Algorithms – Find an Element in 2 dimensional sorted array**

**Objective :** To **Find an Element in 2 dimensional array where rows and columns are sorted respectively.**

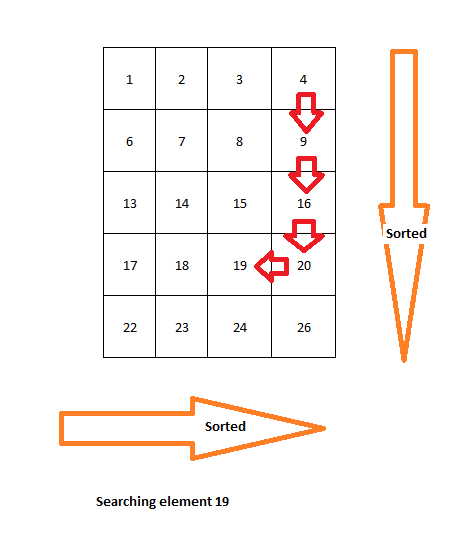
**Input:** A two dimensianl sorted array, arrA[][].



**Output :** True or false based on whether element exists and its location

**Approach:**

* Start from the left top corner, say ele;
* If ele>number -> move left
* If ele<number -> move right
* If you cant move further to find the number , return false



Complete Code:

**package** interviewQuestion;

//Here objective is find an element in two dimensional array

//all rows and columns of an array are sorted in ascending order respectively

**public** **class** FindElementInSorted2DArray {

**public** Boolean findElement(**int** [][] arrA, **int** number){

//start from the left top corner, say ele;

//if ele>number -> move left

//if ele<number -> move right

//if you cant move further to find the number , return false

**int** row = 0;

**int** col = arrA[1].length-1;

**boolean** numberFound = **false**;

System.*out*.print("The Movement : " );

**while**(numberFound==**false**){

**int** ele = arrA[row][col];

System.*out*.print(ele + "->" );

**if**(ele==number){

**return** **true**;

}

**else** **if**(ele>number)col--;

**else** **if** (ele<number)row++;

**if**(row>arrA[0].length-1||col<0)**return** **false**;

}

**return** **false**;

}

**public** **static** **void** main(String args[]){

**int** [][] a = {{1,2,3,4},{6,7,8,9},{13,14,15,16},{17,18,19,20},{22,23,24,26}};

**int** number1 = 15;

**int** number2 = 5;

**int** number3 = 19;

**int** number4 = 25;

FindElementInSorted2DArray f = **new** FindElementInSorted2DArray();

System.*out*.println("The "+ number1 + " present in 2D array a ??? :" + f.findElement(a, number1));

System.*out*.println("The "+ number2 + " present in 2D array a ??? :" + f.findElement(a, number2));

System.*out*.println("The "+ number3 + " present in 2D array a ??? :" + f.findElement(a, number3));

System.*out*.println("The "+ number4 + " present in 2D array a ??? :" + f.findElement(a, number4));

}

}

Output:

The Movement : 4->9->16->15->The 15 present in 2D array a ??? :true

The Movement : 4->9->8->7->6->The 5 present in 2D array a ??? :false

The Movement : 4->9->16->20->19->The 19 present in 2D array a ??? :true

The Movement : 4->9->16->20->The 25 present in 2D array a ??? :false

**Algorithms – Find a Missing Number From a Sequence of Consecutive Numbers**

**Objective :** Find a Missing Number From a Sequence of Consecutive Numbers

**Input:** Array, arrA[] with a missing number and Range

**Output :** missing number

**Approach:**

* Approach is very simple, Add all the given numbers say S
* Calculate sum of N numbers by formula n(n+1)/2 , say N
* Find missing number m = N-S

Example : suppose array given is {1,2,3,4,5,6,8,9,10} and range is 10.

So N will be sum of 1 to 10 = 10(10+1)/2 = 55

S will be sum of all the array elements which is = 48

So missing number will be = 55-48 = 7

**Complete Code:**

**package** interviewQuestion;

//find the missing number from the sequence of consecutive number

//Approach is very simple, Add all the given numbers say S

//Calculate sum of N numbers by formula n(n+1)/2 , say N

//Find missing number m = N-S

**public** **class** FindMissingNumber {

**int** Sum;

**int** Sum\_N;

**public** **int** missingNumber(**int** [] arrA, **int** size){

Sum\_N = size\*(size+1)/2;

**for**(**int** i=0;i<arrA.length;i++){

Sum +=arrA[i];

}

**return** Sum\_N-Sum;

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {1,2,3,4,5,7,8,9,10};

System.*out*.println("Missing number is :" + (**new** FindMissingNumber()).missingNumber(arrA,10));

}

}

Output :

Missing number is :6

**Algorithms – Find two Missing Numbers in a Sequence of Consecutive Numbers**

**Objective :** Find two Missing Numbers in a Sequence of Consecutive Numbers

**Input:** Array, arrA[] with two missing numbers and Range

**Output :** Two missing numbers

**Approach:**

* Approach is very simple, Add all the given numbers say S
* Calculate sum of N numbers by formula n(n+1)/2 , say N
* Find sum of two missing numbers a+b = N-S
* Now take the product of all given numbers say P
* Now take the product of N numbers , say Np;
* Find the product of two missing numbers ab = Np-P
* Now we have a+b and ab , we can easily calculate a and b

**Example :**

Given array : {10,2,3,5,7,8,9,1}; Range : 10

N (Sum of 1 to 10 ) = 55

S (Sum of given elements ) = 45

a+b = 10------------------------------------(1)

Np(Product of 1 to 10) = 3628800

P(Product of given elements) = 151200

So a\*b = 24---------------------------------(2)

Now we have two equations and two variables, if we solve we will get values 6 and 4.

**Complete Code:**

**package** interviewQuestion;

//find the two missing numbers from the sequence of consecutive number

//Approach is very simple, Add all the given numbers say S

//Calculate sum of N numbers by formula n(n+1)/2 , say N

//Find sum of two missing numbers a+b = N-S

//Now take the product of all given numbers say P

//now take the product of N numbers , say Np;

//find the product of two missing numbers ab = Np-P

//now we have a+b and ab , we can easily calculate a and b

**public** **class** FindTwoMissingNumbers {

**int** Sum;

**int** SumN;

**int** P=1;

**int** Np=1;

**int** a,b;

**public** **int** [] missingNumbers(**int** [] arrA, **int** range){

SumN = range\*(range+1)/2;

**for**(**int** i=0;i<arrA.length;i++){

Sum +=arrA[i];

}

**int** s= SumN-Sum;

**for**(**int** i=0;i<arrA.length;i++){

P \*=arrA[i];

}

**for**(**int** i=1;i<=range;i++){

Np \*=i;

}

**int** product = Np/P;

// System.out.println(product);

**int** diffSqr = (**int**)Math.*sqrt*(s\*s-4\*product); // (a-b)^2 = (a+b)^2-4ab

a = (s+diffSqr)/2;

b= s-a;

**int** [] result = {a,b};

**return** result;

}

**public** **static** **void** main(String args[]){

**int** [] arrA = {10,2,3,5,7,8,9,1};

FindTwoMissingNumbers f = **new** FindTwoMissingNumbers();

**int** [] results = f.missingNumbers(arrA, 10);

System.*out*.println("Missing numbers are :" + results[0] + " and " + results[1]);

}

}

**Output:**

Missing numbers are :6 and 4

**Algorithms – Find a peak element in a Given Array**

**Objective :** Find a peak element in a Given Array, where peak element is the one.

**Peak Element:** peak element is the element which is greater than or equal to both of its neighbors.

**Input:** Array, arrA[] .

**Output:** A peak element and its index

**Approach:**

A simple approach is to do a linear scan to a array and using few variables we can find a peak element. But the Time Complexity will be O(n) but real question is, Can we do better?

The Answer is yes, by using Binary Search techniques.

* If middle element is the peak element, return it
* If middle element is smaller than its left element , we will get our peak element on the left half
* If middle element is the smaller than its right element, we will our peak element on the right.

**Notes:**

1. If array has all the same elements, every element is a peak element.
2. Every array has a peak element.
3. Array might have has many peak elements but we are finding only one.
4. If array is in ascending or descending order then last element or the first element of the array will be the peak element respectively.

**Complete Code:**

**package** interviewQuestion;

//we will use binary search techniques

//if middle element is the peak element, return it

//if middle element is smaller than its left element , we will get our peak element on the left half

//if middle element is the smaller than its right element, we will our peak element on the right.

**public** **class** PeakElement {

**public** **int** peak(**int** [] arrA,**int** low, **int** high, **int** size){

**int** mid = (low+high)/2;

**if**((mid==0||arrA[mid]>=arrA[mid-1]) && (arrA[mid]>=arrA[mid+1]||mid==size-1)){

**return** mid;

}

**else** **if**(mid>0 && arrA[mid]<arrA[mid-1]) **return** peak(arrA,low,mid-1,size);

**else** **return** peak(arrA,mid+1,high,size);

}

**public** **static** **void** main(String args[]){

PeakElement pe = **new** PeakElement();

**int** arrA[] = { 1,2,3,4,0,1,5,4,3,2,1};

**int** peakEle = pe.peak(arrA, 0, arrA.length-1, arrA.length);

System.*out*.println("Peak Element is found at index [" + peakEle +"] = "+ arrA[peakEle]);

}

}

**Output:**

Peak Element is found at index [6] = 5

**Algorithms – Find Whether Given String is palindrome or Not.**

**Objective :** Find Whether Given String is palindrome or Not.

**Input:** A String,

**Output:** true or false on whether string is palindrome or not

**Approach:**

* Use recursive approach
* Compare first and last characters if they are not same- return false
* If they are same make, remove the first and last characters and make a recursive call.

**Example:**

Jain niaJ => compare ‘J’ with ‘J’ =>returns true

ain nia => compare ‘a’ with ‘a’ =>returns true

in ni => compare ‘i’ with ‘i’ =>returns true

n n => compare ‘n’ with ‘n’ =>returns true

string length <2 => returns true

**Complete Code:**

**package** interviewQuestion;

//Use recursive approach

//Compare first and last characters if they are not same- return false

//If they are same make, remove the first and last characters and make a recursive call.

**public** **class** Palindrome {

**public** Boolean isPalindrome(String strX){

**if**(strX.length()<2) **return** **true**;

**if**(strX.charAt(0)==strX.charAt(strX.length()-1)){

isPalindrome(strX.substring(0, strX.length()-2));

}

**else** **return** **false**;

**return** **true**;

}

**public** **static** **void** main(String args[]){

String S1 = "Sumit";

String S2 = "SumuS";

String S3 = "ABCDEFGHGFEDCBA";

String S4 = "Jain niaJ";

Palindrome p = **new** Palindrome();

System.*out*.println("Is "+ S1 + " Palindrome ??? :" + p.isPalindrome(S1));

System.*out*.println("Is "+ S2 + " Palindrome ??? :" + p.isPalindrome(S2));

System.*out*.println("Is "+ S3 + " Palindrome ??? :" + p.isPalindrome(S3));

System.*out*.println("Is "+ S4 + " Palindrome ??? :" + p.isPalindrome(S4));

}

}

**Output:**

Is Sumit Palindrome ??? :false

Is SumuS Palindrome ??? :true

Is ABCDEFGHGFEDCBA Palindrome ??? :true

Is Jain niaJ Palindrome ??? :true

**Algorithms – Find Whether Given the Sequence of parentheses are well formed.**

**Objective:** Find Whether Given the Sequence of parentheses are well formed.

**Input:** A String contains a sequence of parentheses

**Output:** true or false on whether parentheses are well formed or not

**Approach:**

* Idea is to have two counters, one for open parentheses '{' and one for close '}'
* Read one character at a time and increment one of the counters
* If any given point of time count of close parentheses is greater than the open one, return false
* If at the end both counters are equal, return true

**Example: { { } { } } – Well formed**

**{ { } { = Not well formed**

**Complete Code:**

**package** interviewQuestion;

**public** **class** WellFormedParentheses {

**public** Boolean isWellFormed(String strParentheses){

**if**(strParentheses==**null**){

**return** **false**;

}

//Idea is to have two counters, one for open parentheses '{' and one for close '}'

//Read one character at a time and increment one of the counters

//If any given point of time count of close parentheses is greater than the open one, return false

//If at the end both counters are equal, return true

**int** openParenCounter=0;

**int** closeParenCounter=0;

**for**(**int** i =0; i<strParentheses.length();i++){

**char** x = strParentheses.charAt(i);

**if**(x=='{') openParenCounter++;

**else** **if**(x=='}') closeParenCounter++;

**if**(closeParenCounter>openParenCounter){

**return** **false**;

}

}

**if**(openParenCounter==closeParenCounter)**return** **true**;

**else** **return** **false**;

}

**public** **static** **void** main(String args[]){

String x1 = "{{{{}}}}{}{}{}{}{}{}{}{}{}{}{{{}}}";

String x2 = "{{{{}}}}{}{}{}{{}{}{}{}{}{}{}{{{}}}";

String x3 = "{}{";

String x4 = "}{";

String x5 = "{{{{{{{{}}}}}}}}";

WellFormedParentheses w = **new** WellFormedParentheses();

System.*out*.println("Is "+ x1 + " well formed ??? :" + w.isWellFormed(x1));

System.*out*.println("Is "+ x2 + " well formed ??? :" + w.isWellFormed(x2));

System.*out*.println("Is "+ x3 + " well formed ??? :" + w.isWellFormed(x3));

System.*out*.println("Is "+ x4 + " well formed ??? :" + w.isWellFormed(x4));

System.*out*.println("Is "+ x5 + " well formed ??? :" + w.isWellFormed(x5));

}

}

**Output**

Is {{{{}}}}{}{}{}{}{}{}{}{}{}{}{{{}}} well formed ??? :true

Is {{{{}}}}{}{}{}{{}{}{}{}{}{}{}{{{}}} well formed ??? :false

Is {}{ well formed ??? :false

Is }{ well formed ??? :false

Is {{{{{{{{}}}}}}}} well formed ??? :true

**Algorithms – Rearrange Positive and Negative Numbers of Array On Each Side in O(nlogn)**

**Objective:** Rearrange Positive and Negative Numbers of an Array so that one side you have positive numbers and other side with negative Integers without changing their respective order.

Example : Input : 1 -2 3 -4 5 -6 7 -8 9 -10

ReArranged Output : -2 -4 -6 -8 -10 1 3 5 7 9

**Input:** An array with positive and negative numbers

**Output:** Modified array with positive numbers and negative numbers are on each side of the array.

**Approach:**

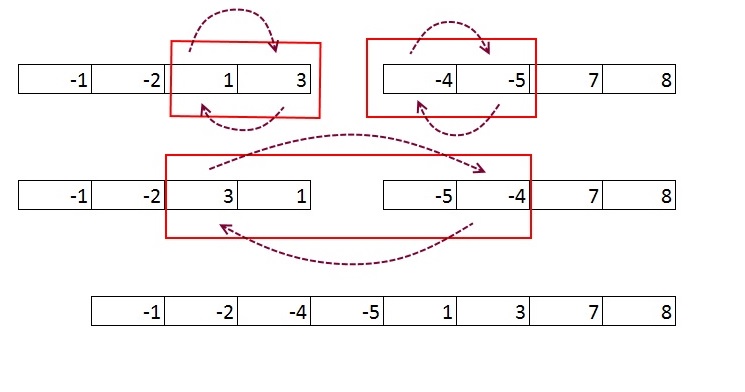
**Method 1.** One naive approach is to have another array of same size and navigate the input array and one scan place all the negative numbers to the new array and in second scan place all the positive numbers to new array. Here the Space complexity will be O(n). We have a better solution which can solve this in O(1) space.

**Method 2: Divide and Conquer**

* This approach is similar to merge sort.
* Divide the array into two half, Solve them individually and merge them.
* Tricky part is in merging.

Merging: (Negative on left side and positives on the right side)

* Navigate the left half of array till you won't find a positive integer and reverse the array after that point.(Say that part is called 'A')
* Navigate the right half of array till you won't find a negative integer and reverse the array after that point. (Say that part is called 'B')
* Now reverse the those parts from both the array (A and B), See example for better explanations



Complete Code:

**package** interviewQuestion;

**public** **class** RearrageArrayPositiveNegative {

**int** [] arrA;

**public** RearrageArrayPositiveNegative(**int** [] arrA){

**this**.arrA = arrA;

}

**public** **void** divideGroups(**int** low, **int** high){

**if**(low>=high) **return**;

**int** mid = (low+high)/2;

divideGroups(low, mid);

divideGroups(mid+1, high);

merge(low,mid,high);

}

**public** **void** merge(**int** low, **int** mid, **int** high){

**int** l = low;

**int** k = mid+1;

**while**(l<=mid && arrA[l]<=0)l++;

**while**(k<=high && arrA[k]<=0)k++;

reverse(l,mid);

reverse(mid+1,k-1);

reverse(l,k-1);

}

**public** **void** reverse(**int** x, **int** y){

**while**(y>x){

**int** temp = arrA[x];

arrA[x]=arrA[y];

arrA[y]=temp;

x++;

y--;

}

}

**public** **void** display(){

**for**(**int** i=0;i<arrA.length;i++){

System.*out*.print(" "+ arrA[i]);

}

}

**public** **static** **void** main(String args[]){

**int** [] a = {1,-2,3,-4,5,-6,7,-8,9,-10};

RearrageArrayPositiveNegative r = **new** RearrageArrayPositiveNegative(a);

System.*out*.print("Input : ");r.display();

r.divideGroups(0, a.length-1);

System.*out*.println("");

System.*out*.print("ReArranged Output : ");r.display();

}

}

Output:

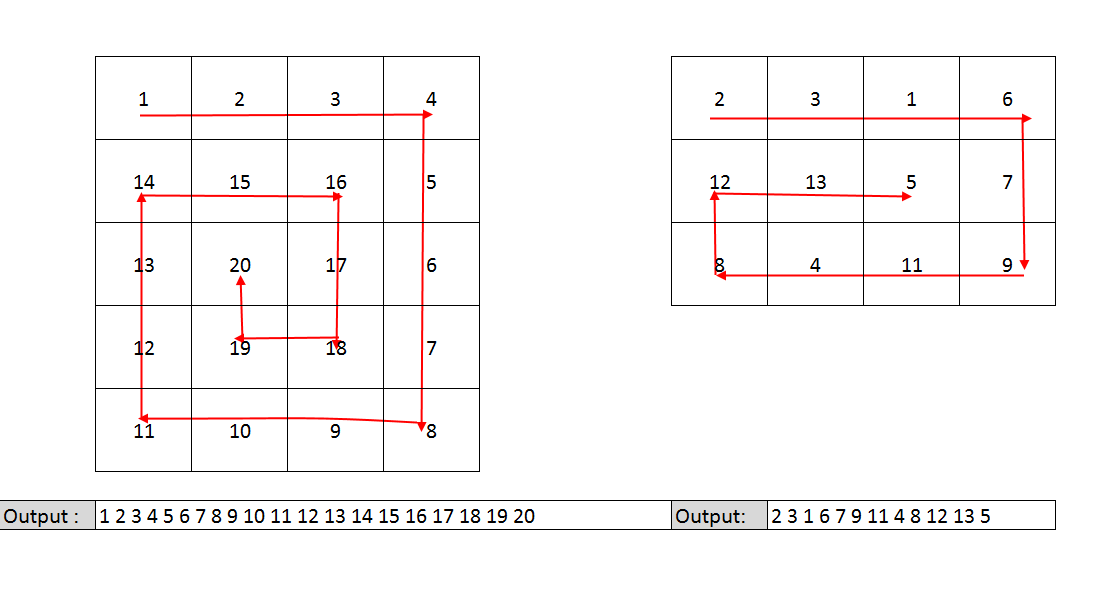
Input : 1 -2 -3 -4 5 -6 7 -8 9 -10 -11 -12 20

ReArranged Output : -2 -3 -4 -6 -8 -10 -11 -12 1 5 7 9 20

**Algorithms – Print All Elements of Two Dimensional Array in Spiral**

**Objective:** Print all the elements of two dimensional array in spiral.

Example :



**Input:** Two dimensional array

**Output:** All array elements printed in spiral.

**Approach:**

* Start printing from first row.
* Print row and columns, forward and backward alternatively
* With every iteration of (either row or column), reduce the size of an row or column by 1
* Call recursively

Complete Code:

**package** interviewQuestion;

**public** **class** Print2DArrayInSpiral {

**public** **int** arrA[][] = { { 1, 2, 3, 4, 5 }, { 18, 19, 20, 21, 6 },

{ 17, 28, 29, 22, 7 }, { 16, 27, 30, 23, 8 },

{ 15, 26, 25, 24, 9 }, { 14, 13, 12, 11, 10 } };

**public** **int** printSpiral(**int** row\_S, **int** row\_E, **int** col\_S, **int** col\_E,**boolean** reverse, **boolean** rowPrint) {

**if** (row\_S > row\_E && col\_S>col\_E) {

**return** 1;

}

**if** (rowPrint == **false**) {

**if** (reverse == **false**) {

**for** (**int** i = col\_S; i <= col\_E; i++) {

System.*out*.print(" " + arrA[row\_S][i]);

}

}

row\_S++;

rowPrint = **true**;

reverse = **false**;

}

**if** (rowPrint == **true**) {

**if** (reverse == **false**) {

**for** (**int** i = row\_S; i <= row\_E; i++) {

System.*out*.print(" " + arrA[i][col\_E]);

}

}

col\_E--;

rowPrint = **false**;

reverse = **true**;

}

**if** (rowPrint == **false**) {

**if** (reverse == **true**) {

**for** (**int** i = col\_E; i >= col\_S; i--) {

System.*out*.print(" " + arrA[row\_E][i]);

}

}

row\_E--;

rowPrint = **true**;

reverse = **true**;

}

**if** (rowPrint == **true**) {

**if** (reverse == **true**) {

**for** (**int** i = row\_E; i >= row\_S; i--) {

System.*out*.print(" " + arrA[i][col\_S]);

}

}

col\_S++;

rowPrint = **false**;

reverse = **false**;

}

printSpiral(row\_S, row\_E, col\_S, col\_E, reverse, rowPrint);

**return** 0;

}

**public** **static** **void** main(String args[]) {

Print2DArrayInSpiral p = **new** Print2DArrayInSpiral();

p.printSpiral(0, 5, 0, 4, **false**, **false**);

}

}

Output:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

**Algorithms – Find Out Whether String Contains All The Unique Characters**

**Objective:** find out whether in a given string contains all the unique characters

**Input:** A String

**Output:** True or false based upon whether string contains all the unique characters or not

**Approach:**

**Method 1.**

**When characters are not ASCII but could be anything alphabets or special characters**

* Create a boolean array of size 256, and put false at every index.
* Navigate the input string one character at a time, say 'char a'
* Check array position of array[a], if it is false, make it true.
* If it is false, update it as true.

**Method 2:**

Sort the array and do the linear scan to find out whether string contains unique elements or not.

**Complete Code:**

**package** interviewQuestion;

//This Program is to find out whether String contains all the unique characters

//With out using any additional data structures

**public** **class** UniqueCharString {

**private** String ip;

**public** UniqueCharString(String ip){

**this**.ip = ip;

}

//method 1 : When characters are not ASCII but could be anything alphabets or special characters

//Time Complexity : O(n)

//Space Complexity : O(1)

//

**public** Boolean UniChars(){

Boolean [] bln = **new** Boolean[256];

**for**(**int** i=0;i<256;i++){

bln[i]=**false**;

}

**for**(**int** i = 0;i<ip.length();i++){

**char** a = ip.charAt(i);

**if**(bln[a]==**true**){

**return** **false**;

}

**else**{

bln[a]=**true**;

}

}

**return** **true**;

}

//method 2: Sort the array and do the linear scan to find out whether string

//contains unique elements or not

//Time Complexity : O(nLogn)

//Space Complexity : O(n)

**public** Boolean UniqueCharSorting(){

**char** [] a = ip.toCharArray();

java.util.Arrays.*sort*(a);

String tmp = **new** String(a);

**for**(**int** i=1;i<tmp.length();i++){

**char** t = tmp.charAt(i-1);

**if**(t==tmp.charAt(i)){

**return** **false**;

}

}

**return** **true**;

}

**public** **static** **void** main(String args[]){

String a = "Sumit\_Jain";

UniqueCharString u = **new** UniqueCharString(a);

System.*out*.println("Method 1 : Does String ' " + a +" ' has all unique characters :" + u.UniChars());

a = "Sumit";

u = **new** UniqueCharString(a);

System.*out*.println("Method 1 : Does String ' " + a +" ' has all unique characters :" + u.UniChars());

a = "Sumit\_Jain";

u = **new** UniqueCharString(a);

System.*out*.println("Method 2 : Does String ' " + a +" ' has all unique characters :" + u.UniqueCharSorting());

a = "Sumit";

u = **new** UniqueCharString(a);

System.*out*.println("Method 2 : Does String ' " + a +" ' has all unique characters :" + u.UniqueCharSorting());

}

}

Output:

Method 1 : Does String ' Sumit\_Jain ' has all unique characters :false

Method 1 : Does String ' Sumit ' has all unique characters :true

Method 2 : Does String ' Sumit\_Jain ' has all unique characters :false

Method 2 : Does String ' Sumit ' has all unique characters :true

**Algorithms – Print All The Permutations Of a String**

**Objective:** find out whether in a given string contains all the unique characters

**Input:** A String

**Output:** Print all the permutations of a string

Example:

Input : abc

**Output:** abc acb bac bca cba cab

**Approach:**

**Algorithms – Quick Sort Implementation**

**Objective:** To sort an array in increasing or decreasing order using Quick Sort.

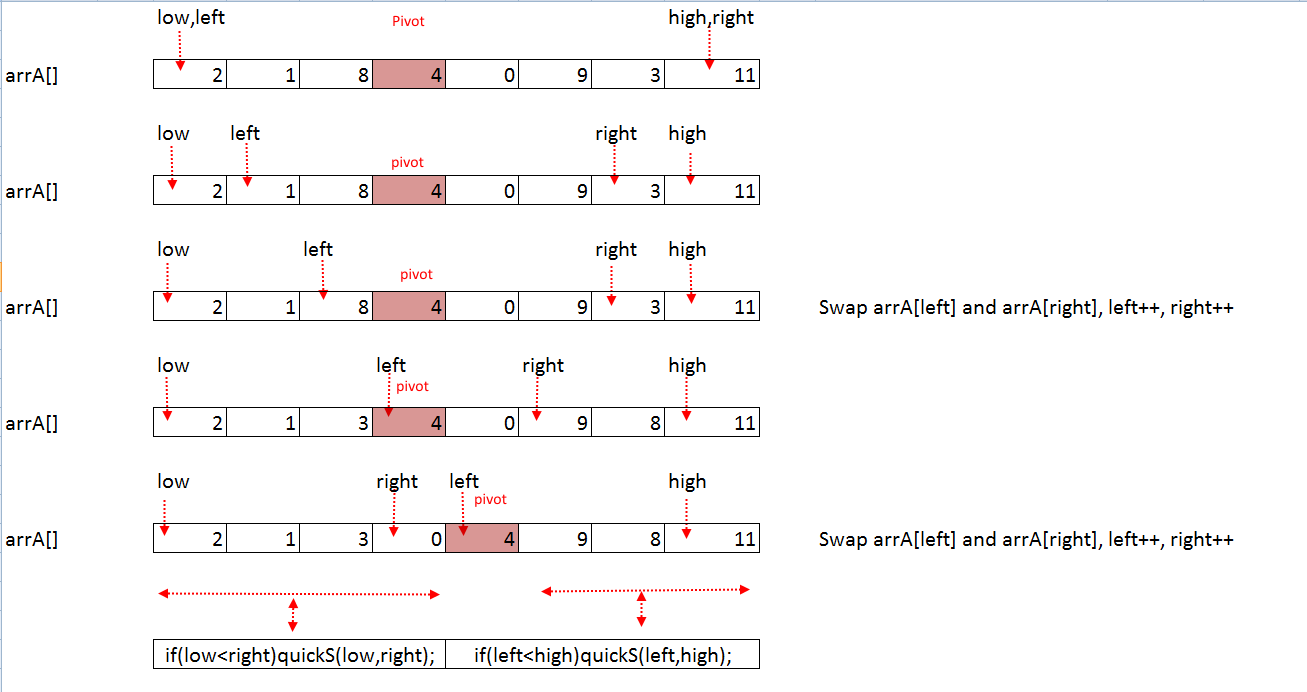
**Input:** An Array arrA[]

**Output:** A sorted array.

**Approach:**

* Choose any element from the array and call it as pivot element, Example here we have selected middle element as pivot
* Place all the elements smaller than pivot in the left side of pivot.
* Place all the elements greater than pivot in the right side of pivot.
* Sort left side and right side recursively.

Example:



**Complete Code:**

**package** interviewQuestion;

**public** **class** QuickSort

{

**private** **int** [] arrA;

**public** QuickSort(**int** [] arrA){

**this**.arrA = arrA;

}

**public** **void** quickS(**int** low, **int** high){

**int** mid = (low+high)/2;

**int** left = low;

**int** right = high;

**int** pivot = arrA[mid]; //select middle element as pivot

**while**(left<=right){

**while**(arrA[left]<pivot) left++;//find element which is greater than pivot

**while**(arrA[right]>pivot)right--;////find element which is smaller than pivot

//System.out.println(arrA[left] + " " + pivot + " " + arrA[right] );

//if we found the element on the left side which is greater than pivot

//and element on the right side which is smaller than pivot

//Swap them, and increase the left and right

**if**(left<=right){

**int** temp = arrA[left];

arrA[left] = arrA[right];

arrA[right]= temp;

left++;

right--;

}

}

//Recursion on left and right of the pivot

**if**(low<right)quickS(low,right);

**if**(left<high)quickS(left,high);

}

**public** **void** display(){

**for**(**int** i =0;i<arrA.length;i++){

System.*out*.print(" " + arrA[i] );

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

// your code goes here

**int** a [] = { 2,1,8,4,0,9,3,11};

QuickSort i = **new** QuickSort(a);

System.*out*.print("UnSorted : ");

i.display();

i.quickS(0,a.length-1);

System.*out*.println("");

System.*out*.print("Quick Sorted : ");

i.display();

}

}

Output:

UnSorted : 2 1 8 4 0 9 3 11

Quick Sorted : 0 1 2 3 4 8 9 11

Time Complexity : O(n^2) worst case and O(nlogn) average case analysis

Space Complexity : O(1)

**Algorithms – Given an array and a number k, check for pair in array with sum as k in O(nlgn)**

**Keywords :** **Given an array A[] and a number x, check for pair in A[] with sum as x, Given an array and a number k, check for pair in array with sum as k, Checking if there are 2 elements in an array that sum to X in O(n lg n),** **Find two numbers in an array whose sum is x,** **Find a pair of elements from an array whose sum equals a given number**

**Objective:** To find out whether in a given array there exists or not two numbers whose sum is exactly equals to a given number

**Input:** An array arrA[], A number k

**Output:** True or false based upon we have found any two numbers in array arrA[] whose sum is equal to k

**Approach:**

Method 1: Using Binary Search

* First sort the array using Merge Sort(To know about Merge Sort and its implementation Click Here)
* Now do the linear scan to the from the left array , say starting index i=0
* Calculate Rem\_Sum = number - arrA[i]
* If Rem\_Sum<0, move to the next element
* If Rem\_Sum>0, Perform Binary Search for Rem\_Sum on the remaining elements on the right side.

Time Complexity - O(nlogn)

**Method 2: Using Both Ends**

* First sort the array using Merge Sort(To know about Merge Sort and its implementation Click Here)
* Start from the both ends of the array
* Add first (say 'a') and last element(say 'b') of the array say S
* If S > number , S = S-(last\_element) + (element before that)
* If S < number , S = S - (first element) + (next element)
* If if S=number, return true
* Repeat step
* If iteration gets over and retrun false.

**Complete Code:**

**package** interviewQuestion;

**public** **class** TwoNumbersInArray {

**private** **int** [] arrA;

**private** **int** number;

**public** TwoNumbersInArray(**int** [] arrA,**int** number){

**this**.arrA = arrA;

**this**.number = number;

}

**public** Boolean usingBinarySearch(){

//1. First sort the array

MergeSort m = **new** MergeSort(arrA);

**int** [] arrSorted = m.mergeSorting();

BinarySearch bs = **new** BinarySearch(arrSorted);

//2. now do the linear scan to the from the left array , say starting index i=0

//3. Calculate Rem\_Sum = number - a[i]

//4. if Rem\_Sum<0, move to the next element

//5. if Rem\_Sum>0, Perform Binary Search on the remaining elements on the right side.

**for**(**int** i =0;i<arrA.length-1;i++){

**int** RemS = number - arrA[i];

**if**(RemS>0){

**if**(bs.Search(i+1, arrA.length-1, RemS)) **return** **true**;

}

}

**return** **false**;

}

**public** Boolean usingBothEnds(){

//1. First sort the array

MergeSort m = **new** MergeSort(arrA);

**int** [] arrSorted = m.mergeSorting();

//2. Start from the both ends of the array

//3. add first (say 'a') and last element(say 'b') of the array say S

//4. if S > number , S = S-(last\_element) + (element before that)

//5. if S < number , S = S - (first element) + (next element)

//6. if S=number, return true

//7. Repeat step

//8. If iteration gets over and retrun false.

**int** i =0;

**int** j = arrSorted.length-1;

**int** Sum = 0;

**while** (i!=j){

Sum = arrSorted[i]+ arrSorted[j];

**if**(Sum==number)**return** **true**;

**else** **if** (Sum<number)i++;

**else** **if** (Sum>number)j--;

}

**return** **false**;

}

**public** **static** **void** main(String[] args){

**int** a [] = { 1,2,3,4,5,16,17,18,19,249};

**int** number = 269;

**int** number1 = 8;

TwoNumbersInArray tn = **new** TwoNumbersInArray(a, number);

System.*out*.println("USING Both Ends -Sum of two numbers in array A is "+ number + " ??? :" + tn.usingBothEnds());

System.*out*.println("USING Binary Search -Sum of two numbers in array A is "+ number + " ??? :" + tn.usingBinarySearch());

TwoNumbersInArray tn1 = **new** TwoNumbersInArray(a, number1);

System.*out*.println("USING Both Ends -Sum of two numbers in array A is "+ number1 + " ??? :" + tn1.usingBothEnds());

System.*out*.println("USING Binary Search -Sum of two numbers in array A is "+ number1 + " ??? :" + tn1.usingBinarySearch());

}

}

Output:

USING Both Ends -Sum of two numbers in array A is 269 ??? :false

USING Binary Search -Sum of two numbers in array A is 269 ??? :false

USING Both Ends -Sum of two numbers in array A is 8 ??? :true

USING Binary Search -Sum of two numbers in array A is 8 ??? :true

**Algorithms - Find The Longest Sequence Of Prefix Shared By All The Words In A String**

**Objective:** Find The Longest Sequence Of Prefix Shared By All The Words In A String

**Input:** A String

**Output:** The longest sequence of prefix common in all the words in a string

**Example:**

"Bedroom BedClock BedTable BedWall" => "Bed"

**Approach:**

* Split the input by blank space and store it in arrA[].
* Create int resultLen and store the first index string length in it (int resultLen = arrA[0].length())
* Create another interger variable, int curr
* Now run a loop in rest of the array.
* Check if curr < resultLen and curr<length of current string in a loop
* If so check if character at curr position matched in first index string and with the current string a loop, if so, increase curr by 1
* Change resultLen = curr
* Return substring of resultLen length

**Complete Code:**

**package** interviewQuestion;

**public** **class** LongestPrefixSequence {

**private** String [] arrA;

**public** LongestPrefixSequence(String [] arrA){

**this**.arrA = arrA;

}

**public** String findPrefix(){

**int** resultLen = arrA[0].length();

**int** curr;

**for**(**int** i=1;i<arrA.length;i++){

curr=0;

**while**(curr<resultLen && curr<arrA[i].length() && arrA[0].charAt(curr)==arrA[1].charAt(curr)){

curr++;

}

resultLen = curr;

}

**return** arrA[0].substring(0,resultLen);

}

**public** **static** **void** main(String args[]){

String x = "Sumit Summation Summit Sum";

String [] arrA = x.split(" ");

LongestPrefixSequence lp = **new** LongestPrefixSequence(arrA);

System.*out*.println("Original String : " + x);

System.*out*.println("Common Prefix is : " + lp.findPrefix());

}

}

**Output:**

Original String : Sumit Summation Summit Sum

Common Prefix is : Sum

**Keyword:** Find The Longest Sequence Of Prefix Shared By All The Words In A String, The longest sequence of prefix common in all the words in a string, The longest substring of prefix common in all the words in a string**, Google interview, Common prefix in all strings, Common prefix in multiple strings**

**Algorithms - Singly Linked List Implementation**

Linked List- As the name suggests it's a list which is linked.

* Linked List consist of Nodes
* Nodes are nothing but objects of a class and each node has data and a link to the next node.

**class** Node {

**public** **int** data;

**public** Node next;

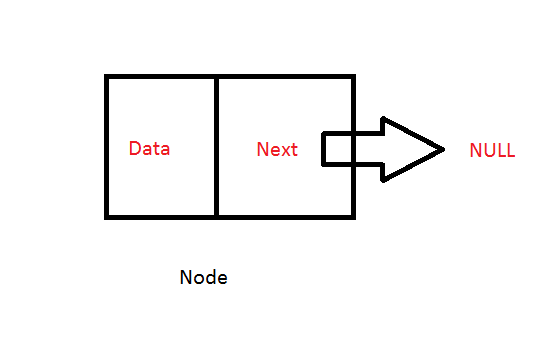
**public** Node(**int** data) {

**this**.data = data;

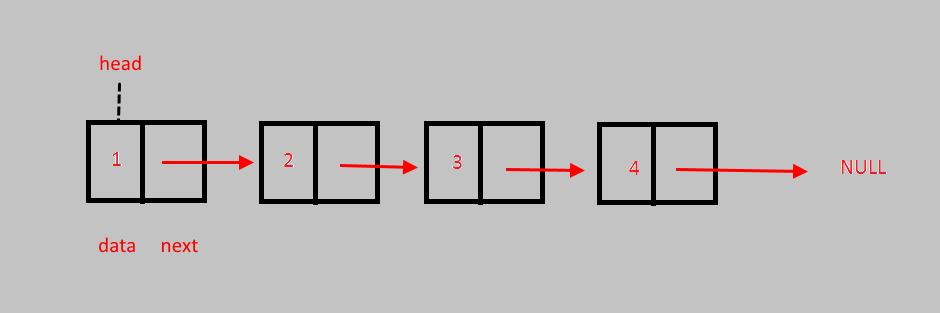
**this**.next = **null**;

}

}



* The last node in the list points to NULL , so when you reach there you will know that the list ends here.



**Operations:**

**Add at the Start :** Add a node the beginning of the linked list. Its O(1).

**Add at the End :** Add a node at the end of the linked list. its O(n) since to add a node at the end you need to go till the end of the array.

**Delete at the Start :** Delete a node from beginning of the linked list. Its O(1).

**Delete at the End :** Delete a node from the end of the linked list. its O(n) since to delete a node at the end you need to go till the end of the array.

**Get Size:** returns the size of the linked list.

**Get Element at Index :** Return the element at specific index, if index is greater than the size then return -1. its O(n) in worst case.

**Add Element at Specific Index :** Add element at specific index. If index is greater than size then print "INVALID POSITION". Worst case its O(n)

Display(): Prints the entire linked list. O(n).

**Complete Code:**

**package** interviewQuestion;

**public** **class** LinkListImplementation {

**public** **static** **void** main(String[] args) **throws** java.lang.Exception {

LinkedListT a = **new** LinkedListT();

a.addAtBegin(5);

a.addAtBegin(15);

a.addAtEnd(20);

a.addAtEnd(21);

a.deleteAtBegin();

a.deleteAtEnd();

a.addAtIndex(10, 2);

a.addAtEnd(15);

a.display();

System.*out*.println("\n Size of the list is: " + a.size);

System.*out*.println(" Element at 2nd position : " + a.elementAt(2));

System.*out*.println(" Searching element 20, location : " + a.search(15));

}

}

**class** Node {

**public** **int** data;

**public** Node next;

**public** Node(**int** data) {

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListT {

**public** Node head;

**public** **int** size;

**public** LinkedListT() {

head = **null**;

}

**public** **void** addAtBegin(**int** data) {

Node n = **new** Node(data);

n.next = head;

head = n;

size++;

}

**public** **int** deleteAtBegin() {

**int** tmp = head.data;

head = head.next;

size--;

**return** tmp;

}

**public** **void** deleteAtEnd() {

Node currNode = head;

**if** (head.next == **null**) {

head = **null**;

} **else** {

**while** (currNode.next.next != **null**) {

currNode = currNode.next;

}

**int** temp = currNode.next.data;

currNode.next = **null**;

size--;

}

}

**public** **void** addAtEnd(**int** data) {

**if** (head == **null**) {

addAtBegin(data);

} **else** {

Node n = **new** Node(data);

Node currNode = head;

**while** (currNode.next != **null**) {

currNode = currNode.next;

}

currNode.next = n;

size++;

}

}

**public** **int** elementAt(**int** index){

**if**(index>size){

**return** -1;

}

Node n = head;

**while**(index-1!=0){

n=n.next;

index--;

}

**return** n.data;

}

**public** **int** getSize(){

**return** size;

}

**public** **int** search(**int** data){

Node n = head;

**int** count = 1;

**while**(n!=**null**){

**if**(n.data==data){

**return** count;

}**else**{

n = n.next;

count++;

}

}

**return** -1;

}

**public** **void** addAtIndex(**int** data, **int** position){

**if**(position == 1){

addAtBegin(data);

}

**int** len = size;

**if** (position>len+1 || position <1){

System.*out*.println("\nINVALID POSITION");

}

**if**(position==len+1){

addAtEnd(data);

}

**if**(position<=len && position >1){

Node n = **new** Node(data);

Node currNode = head; //so index is already 1

**while**((position-2)>0){

System.*out*.println(currNode.data);

currNode=currNode.next;

position--;

}

n.next = currNode.next;

currNode.next = n;

size++;

}

}

**public** **void** display() {

System.*out*.println("");

Node currNode = head;

**while** (currNode != **null**) {

System.*out*.print("->" + currNode.data);

currNode = currNode.next;

}

}

}

**Output:**

->5->10->20->15

Size of the list is: 4

Element at 2nd position : 10

Searching element 20, location : 4

**Algorithms - Merge or Combine Two Sorted Linked Lists**

**Objective:** Given two sorted linked lists, objective is to merge both the lists in sorted order.

**Input:** Two sorted linked list List a, List b.

**Example:**

**List a :** ->2->6->18

**List b:** ->1->3->17->19

**Merged List: ->1->2->3->6->17->18->19**

**Approach:**

**Without Recursion:**

* Create a new node say result
* Navigate through both the linked lists at the same time, starting from head
* Compare the first node values of both the linked lists
* Whichever is smaller, add it to the result node
* Move the head pointer of the linked list whose value was smaller
* Again compare the node values
* Keep doing until one list gets over
* Copy the rest of the nodes of unfinished list to the result

**With Recursion:**

* Base Case :

If List A gets finished , return List B.

If List B gets finished, return List A.

* Create a result node and initialize it as NULL
* Check which node (List A or List B) has a smaller value.
* Whichever is smaller, add it to the result node.
* Make recursive call and add the return node as result.next

result.next = recurrsionMerge(nA.next, nB)

**Complete Code:**

**package** interviewQuestion;

//WithOut Recursion

//create a new node say result

//navigate through both the linked lists at the same time, starting from head

//compare the first node values of both the linked lists

//which ever is smaller, add it to the result node

//move the head pointer of the linked list whose value was smaller

//again compare the node values

//keep doing until one list gets over

//copy the rest of the nodes of unfinished list to the result

**public** **class** MergeTwoLinkList {

**private** LinkedListT a;

**private** LinkedListT b;

**public** MergeTwoLinkList(LinkedListT a, LinkedListT b){

**this**.a=a;

**this**.b=b;

}

**public** LinkedListT mergeWithOutRecur(){

LinkedListT result = **new** LinkedListT();

**while**(a.head!=**null** && b.head!=**null**){

// System.out.println(a.head.data + " " + b.head.data);

**if**(a.head.data<b.head.data){

result.addAtEnd(a.head.data);

a.head = a.head.next;

}

**else**{

result.addAtEnd(b.head.data);

b.head = b.head.next;

}

}

**while**(a.head!=**null**){

result.addAtEnd(a.head.data);

a.head = a.head.next;

}

**while**(b.head!=**null**){

result.addAtEnd(b.head.data);

b.head = b.head.next;

}

**return** result;

}

**public** Node recurrsionMerge(Node nA, Node nB){

//base case

Node result = **null**;

**if**(nA==**null**) **return** nB;

**else** **if**(nB==**null**) **return** nA;

**if**(nA.data<nB.data){//Check which node has a smaller value

result = nA; //add it to the result node

result.next = recurrsionMerge(nA.next, nB);//Recursive call and add the return node as result.next

}

**else**{

result = nB; //add it to the result node

result.next = recurrsionMerge(nA, nB.next); //Recursive call and add the return node as result.next

}

**return** result;

}

**public** **void** display(Node head) {

System.*out*.println("");

Node currNode = head;

**while** (currNode != **null**) {

System.*out*.print("->" + currNode.data);

currNode = currNode.next;

}

System.*out*.println("");

}

**public** **static** **void** main(String [] args){

System.*out*.println("Method : with Recursion");

LinkedListT a = **new** LinkedListT();

a.addAtBegin(8);a.addAtBegin(6);a.addAtBegin(5);

LinkedListT b = **new** LinkedListT();

b.addAtBegin(9);b.addAtBegin(7);b.addAtBegin(3);b.addAtBegin(1);

MergeTwoLinkList m = **new** MergeTwoLinkList(a, b);

m.display(a.head);m.display(b.head);

Node result;

result = m.recurrsionMerge(a.head, b.head);

m.display(result);

//method 2

System.*out*.println("Method : without Recursion");

LinkedListT a1 = **new** LinkedListT();

a1.addAtBegin(18);a1.addAtBegin(6);a1.addAtBegin(2);

LinkedListT b1 = **new** LinkedListT();

b1.addAtBegin(19);b1.addAtBegin(17);b1.addAtBegin(3);b1.addAtBegin(1);

MergeTwoLinkList m1 = **new** MergeTwoLinkList(a1, b1);

m1.display(a1.head);m1.display(b1.head);

LinkedListT res = m1.mergeWithOutRecur();

m1.display(res.head);

}

}

**Output:**

Method : with Recursion

->5->6->8

->1->3->7->9

->1->3->5->6->7->8->9

Method : without Recursion

->2->6->18

->1->3->17->19

->1->2->3->6->17->18->19

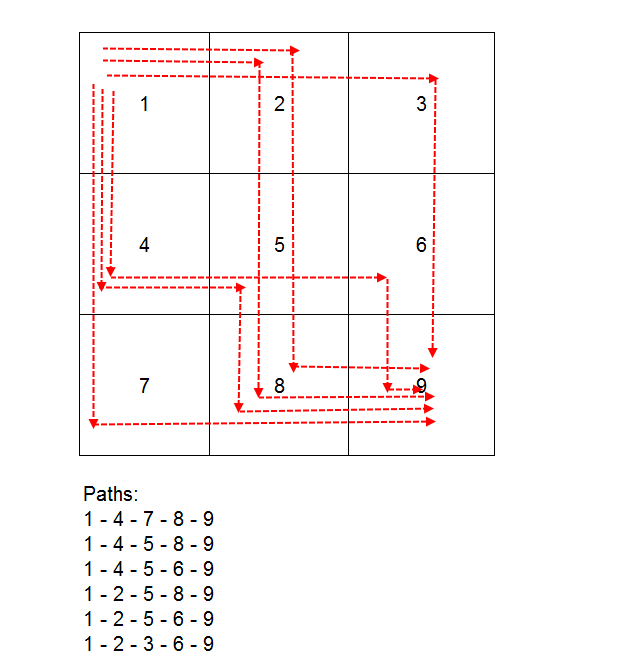
**Algorithms - Print All Paths from Top left to bottom right in Two Dimensional Array**

**Objective:** Print all the paths from left top corner to right bottom corner in two dimensional array.

**Input:** Two Dimensional array

**Output:** Print all the paths.

**Example:**

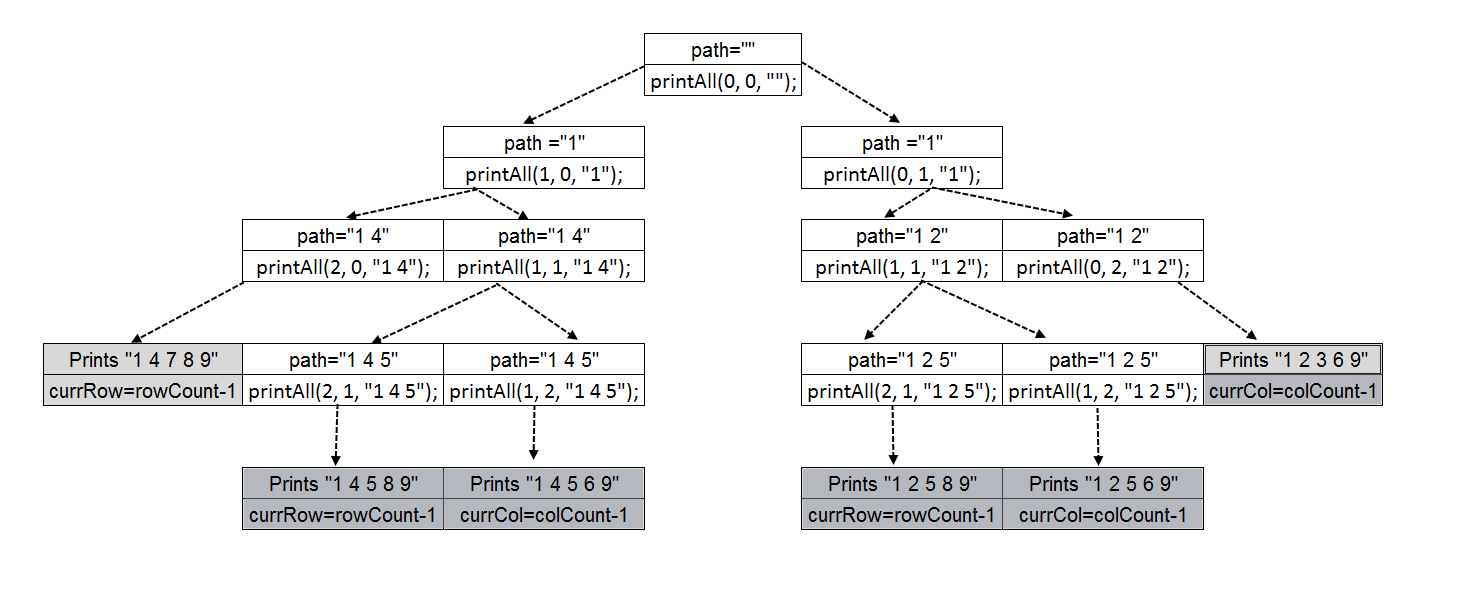


**Approach:**

**As we need to explore all the paths from top left corner to bottom right corner, we will either travel down OR travel right. so every time either we increase the row or column.**

* Recursion is the key here.
* Take the rows count and column counts say rowCount and colCount respectively
* Take currentRow =0 and currentColumn =0 and path =""
* Call printAll(currentRow, currentcolumn,path)
* Add array[0][0] to the path.
* Call recursively printAll(currentRow+1, currentcolumn,path)
* Call recursively printAll(currentRow, currentcolumn+1,path)
* **Base Case 1:** when currentRow = rowCount-1(because array index starts with 0) , print the rest of the columns, return
* **Base Case 2:** when currentcolumn = colCount-1(because array index starts with 0) , print the rest of the rows, return

**Like always you need to trust Recursion to get you the correct result. :)**

****

**Complete Code:**

**package** interviewQuestion;

**public** **class** PrintAllPathIn2DArray {

**int** rowCount;

**int** colCount;

**int** [][] arrA;

**public** PrintAllPathIn2DArray(**int** arrA[][]){

**this**.arrA = arrA;

rowCount=arrA[0].length;

colCount = arrA[1].length;

}

**public** **void** printAll(**int** currentRow, **int** currentColumn, String path){

**if**(currentRow==rowCount-1){

**for**(**int** i=currentColumn;i<colCount;i++){

path += "-" + arrA[currentRow][i];

}

System.*out*.println(path);

**return**;

}

**if**(currentColumn==colCount-1){

**for**(**int** i=currentRow;i<=rowCount-1;i++){

path += "-" + arrA[i][currentColumn];

}

System.*out*.println(path);

**return**;

}

path = path +"-"+ arrA[currentRow][currentColumn];

printAll(currentRow+1,currentColumn,path);

printAll(currentRow,currentColumn+1,path);

}

**public** **static** **void** main(String args[]){

**int** [][]a = {{1,2,3},{4,5,6},{7,8,9}};

PrintAllPathIn2DArray p = **new** PrintAllPathIn2DArray(a);

p.printAll(0, 0, "");

}

}

**Output:**

-1-4-7-8-9

-1-4-5-8-9

-1-4-5-6-9

-1-2-5-8-9

-1-2-5-6-9

-1-2-3-6-9

**Algorithms - Count All Paths from Top left to bottom right in Two Dimensional Array including Diagonal Paths**

**Objective:** Count all the paths from left top corner to right bottom corner in two dimensional array.

**Input:** Two Dimensional array

**Output:** No of paths.

**Approach :**

**1. Recursive**

Recursive solution to this problem is similar to "link to print all paths"

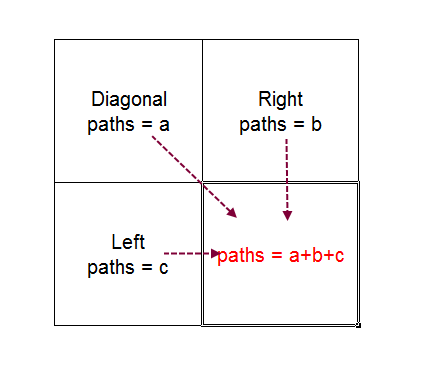
But the Time complexity will be exponential because there will be many sub problems which will be solved again and again to get the final solution. read this : "Dynamic programming vs Recursion

**2. Dynamic Programming (Better Solution)**

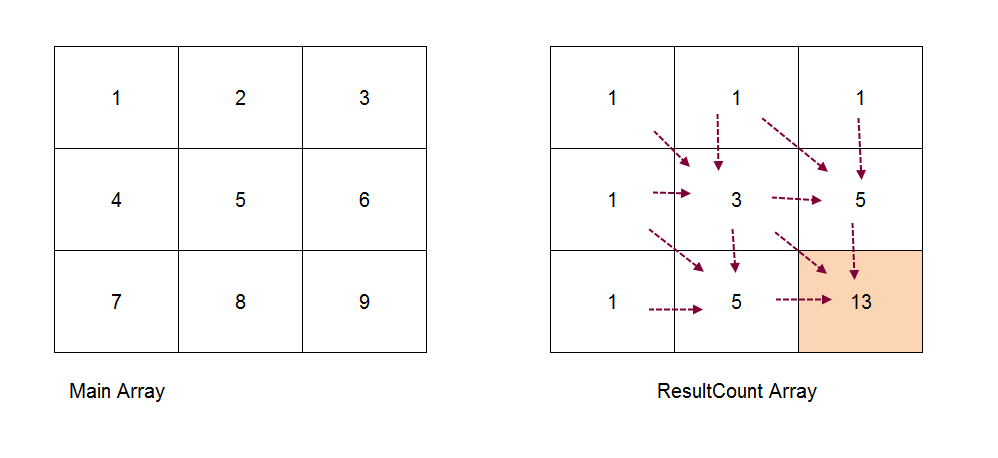
**Create two dimensional resultCount array to store the number of paths from top left corner.**

**Base Case:** To reach to any cell in either first row or column from first cell(top left at 0,0) will be 1.

You can reach to any cell from 3 different ways, from left, from top, from diagonal. So total no of paths to reach to that cell will be sum of all the paths to reach to left, top and diagonal cell. see picture



**Example:**



**Complete Code:**

**package** newInterviewQuestions;

**public** **class** CountAllPaths {

**int** rowCount;

**int** colCount;

**int** [][] arrA;

**public** CountAllPaths(**int** arrA[][]){

**this**.arrA = arrA;

rowCount=arrA.length;

colCount = arrA[0].length;

}

**public** **int** countAllResursion(**int** currentRow, **int** currentColumn){

**if**(currentRow==rowCount-1){

**return** 1;

}

**if**(currentColumn==colCount-1){

**return** 1;

}

**return** countAllResursion(currentRow+1,currentColumn) + countAllResursion(currentRow,currentColumn+1)+ countAllResursion(currentRow+1,currentColumn+1);

}

**public** **int** printAllDynamic(**int** [][] arrA){

**int** [][]resultCount = **new** **int** [arrA.length][arrA[0].length];

**for**(**int** i=0;i<arrA.length;i++){

resultCount[i][0]=1;

}

**for**(**int** i=0;i<arrA[1].length;i++){

resultCount[0][i]=1;

}

**for**(**int** i=1;i<arrA.length;i++){

**for**(**int** j=1;j<arrA[1].length;j++){

resultCount[i][j]= resultCount[i][j-1] + resultCount[i-1][j] + resultCount[i-1][j-1];

}

}

**return** resultCount[arrA.length-1][arrA[0].length-1];

}

**public** **static** **void** main(String args[]){

**int** [][]a = {{1,2,3},{4,5,6},{7,8,9}};

CountAllPaths p = **new** CountAllPaths(a);

System.*out*.println("No of Paths By Recursion: " +p.countAllResursion(0, 0));

System.*out*.println("No of paths By Dynamic Programming: " +p.printAllDynamic(a));

}

}

**Output:**

No of Paths By Recursion: 13

No of paths By Dynamic Programming: 13

**Algorithms - Reverse a Linked List**

**Objective:** Reverse the given linked list.

**Input:** A Linked List

**Output:** Reversed Linked List

**Example:**

Input : ->30->25->20->15->10->5

Reversed : ->5->10->15->20->25->30

**Approach:**

**Iterative:**

* Create 3 nodes, currNode, PrevNode and nextNode.
* Initialize them as currNode = head;,nextNode = **null**;prevNode = **null**;
* Now keep reversing the pointers one by one till currNode!=null.

**while**(currNode!=**null**){

nextNode = currNode.next;

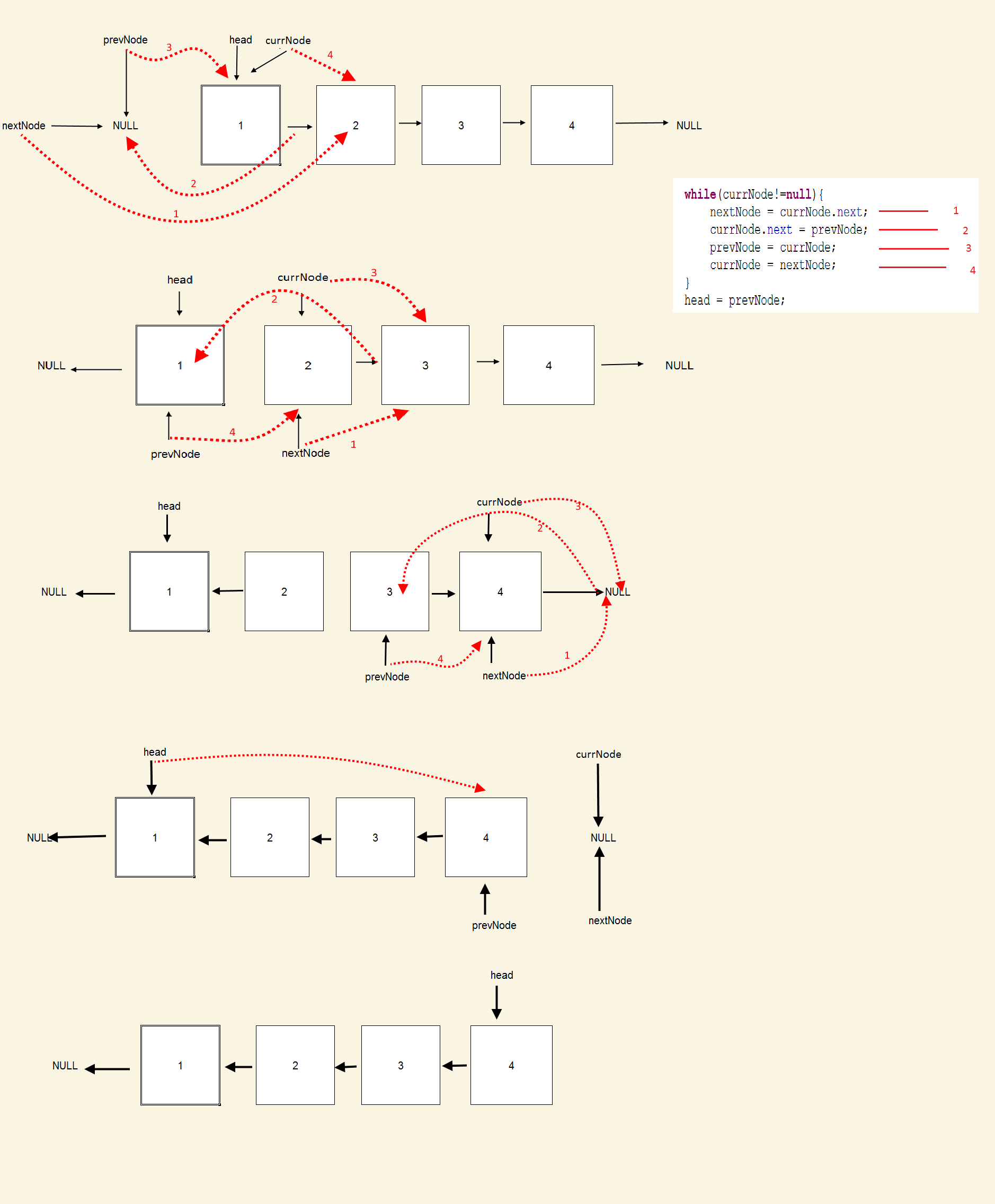
currNode.next = prevNode;

prevNode = currNode;

currNode = nextNode;

}

* At the end set head = prevNode;
* See Example:



**Recursive Approach:**

* Take 3 nodes as Node ptrOne,Node ptrTwo, Node prevNode
* Initialize them as ptrOne = head; ptrTwo=head.next, prevNode = null.
* Call reverseRecursion(head,head.next,null)
* Reverse the ptrOne and ptrTwo
* Make a recursive call for reverseRecursion(ptrOne.next,ptrTwo.next,null)

Complete Code:

**package** reverseALinkedList;

**public** **class** ReverseLinkedList {

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

LinkedListT a = **new** LinkedListT();

a.addAtBegin(5);

a.addAtBegin(10);

a.addAtBegin(15);

a.addAtBegin(20);

a.addAtBegin(25);

a.addAtBegin(30);

// System.out.print("Original Link List 1 : ");

a.display(a.head);

a.reverseIterative(a.head);

LinkedListT b = **new** LinkedListT();

b.addAtBegin(31);

b.addAtBegin(32);

b.addAtBegin(33);

b.addAtBegin(34);

b.addAtBegin(35);

b.addAtBegin(36);

System.*out*.println("");

System.*out*.println("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");

System.*out*.print("Original Link List 2 : ");

b.display(b.head);

b.reverseRecursion(b.head,b.head.next,**null**);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListT{

**public** Node head;

**public** LinkedListT(){

head=**null**;

}

**public** **void** addAtBegin(**int** data){

Node n = **new** Node(data);

n.next = head;

head = n;

}

**public** **void** reverseIterative(Node head){

Node currNode = head;

Node nextNode = **null**;

Node prevNode = **null**;

**while**(currNode!=**null**){

nextNode = currNode.next;

currNode.next = prevNode;

prevNode = currNode;

currNode = nextNode;

}

head = prevNode;

System.*out*.println("\n Reverse Through Iteration");

display(head);

}

**public** **void** reverseRecursion(Node ptrOne,Node ptrTwo, Node prevNode){

**if**(ptrTwo!=**null**){

**if**(ptrTwo.next!=**null**){

Node t1 = ptrTwo;

Node t2 = ptrTwo.next;

ptrOne.next = prevNode;

prevNode = ptrOne;

reverseRecursion(t1,t2, prevNode);

}

**else**{

ptrTwo.next = ptrOne;

ptrOne.next = prevNode;

System.*out*.println("\n Reverse Through Recursion");

display(ptrTwo);

}

}

**else** **if**(ptrOne!=**null**){

System.*out*.println("\n Reverse Through Recursion");

display(ptrOne);

}

}

**public** **void** display(Node head){

//

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

}

Output:

->30->25->20->15->10->5

Reverse Through Iteration

->5->10->15->20->25->30

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Original Link List 2 : ->36->35->34->33->32->31

Reverse Through Recursion

->31->32->33->34->35->36

**Algorithms - Find the Loop in a Linked list, find its length and Break the Loop**

**Objective:** In a given linked list, check whether it contains the loop in it, if yes then find the Loop length and break the loop.

Loop in a linked list means the last node does not point to the null, instead it points to some node in the list.

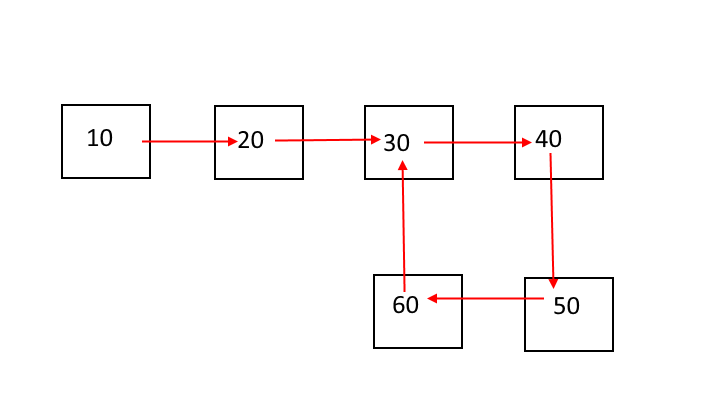
**Input:** A Linked List

**Output:** Linked list contains loop or not, if yes its length and linked list after breaking the loop.

**Example:**

Input : ->10->20->30->40->50->60->30->40->50->60->30->40->50->60->30

here you can see that 30->40->50->60 are repeating ,that means it has a loop



**Approach:**

* **Find the Loop**
* **Take two pointers, both starts from head**
* Move one pointer with normal speed and another with double speed
* If both pointers meets at some point, we have found the loop
* Now find the loop length
* At the point where both pointers have met, stop one pointer and keep moving the nother one
* When another pointer meets the first pointer, stop.
* Keep counting number of hops, that will your loop length
* Now To break the loop
* Move one pointer by the loop length
* Now move both pointers with normal speed.
* When secondpointer.next = first pointer, set secondpinter.next=null.

**Complete Code:**

**package** findLoopInLinkedList;

//Find the Loop

//Take two pointers, both starts from head

//move one pointer with normal speed and another with double speed

//if both pointers meets at some point, we have found the loop

//Now find the loop length

//at the point where both pointers have met, stop one pointer and keep moving the nother one

//when another pointer meets the first pointer, stop.

//keep counting number of hops, that will your loop length

//Now To break the loop

//move one pointer by the loop length

//now move both pointers with normal speed.

//when secondpointer.next = first pointer, set secondpinter.next=null.

**public** **class** FindLoopInLinkedList

{ **public** LinkedListLoop a;

**public** **int** loopLength;

**public** **void** createLoop(){

a = **new** LinkedListLoop();

a.addAtEnd(10);

a.addAtEnd(20);

a.addAtEnd(30);

a.addAtEnd(40);

a.addAtEnd(50);

a.addAtEnd(60);

a.insertLoop(2);

a.displayLoop();

}

**public** **void** findLoop(){

Node ptrOne =a.head;

Node ptrTwo =a.head.next.next;

**while**(ptrOne!=**null**){

**if**(ptrOne!=ptrTwo){

ptrOne = ptrOne.next;

ptrTwo = ptrTwo.next.next;

}

**else**{

System.*out*.println("");

System.*out*.println("Loop Found--starts at " + ptrOne.data);

findLoopLength(ptrOne, ptrTwo);

breakLoop(ptrOne, ptrTwo);

**break**;

}

}

}

**public** **void** findLoopLength(Node one, Node two){

one = one.next;

loopLength = 1;

**while**(one!=two){

one = one.next;

loopLength++;

}

System.*out*.println("Loop length is " + loopLength);

}

**public** **void** breakLoop(Node one, Node two){

one = one.next;

**while**(one.next!=two){

one = one.next;

}

one.next = **null**;

System.*out*.println("Loop breaks");

a.display();

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

FindLoopInLinkedList i = **new** FindLoopInLinkedList();

i.createLoop();

i.findLoop();

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListLoop{

**public** Node head;

**public** LinkedListLoop(){

head=**null**;

}

**public** **void** addAtEnd(**int** data){

Node n = **new** Node(data);

**if** (head==**null**){

n.next = head;

head = n;

}

**else**{

Node currNode = head;

**while**(currNode.next!=**null**){

//System.out.print("---->" + currNode.data);

currNode = currNode.next;

}

currNode.next = n;

}

}

**public** **void** insertLoop(**int** index){

Node endNode = head;

**while**(endNode.next!=**null**){

//System.out.print("---->" + currNode.data);

endNode = endNode.next;

}

Node indexNode = head;

**while**(index!=0){

indexNode = indexNode.next;

index--;

}

endNode.next = indexNode;

}

**public** **void** displayLoop(){

System.*out*.println("");

Node currNode = head;

**int** cnt = 15;

**while**(cnt!=0){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

cnt--;

}

}

**public** **void** display(){

System.*out*.println("");

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

}

Output:

->10->20->30->40->50->60->30->40->50->60->30->40->50->60->30

Loop Found--starts at 30

Loop length is 4

Loop breaks

->10->20->30->40->50->60

**Algorithms - Find Intersection Point in Two Linked List**

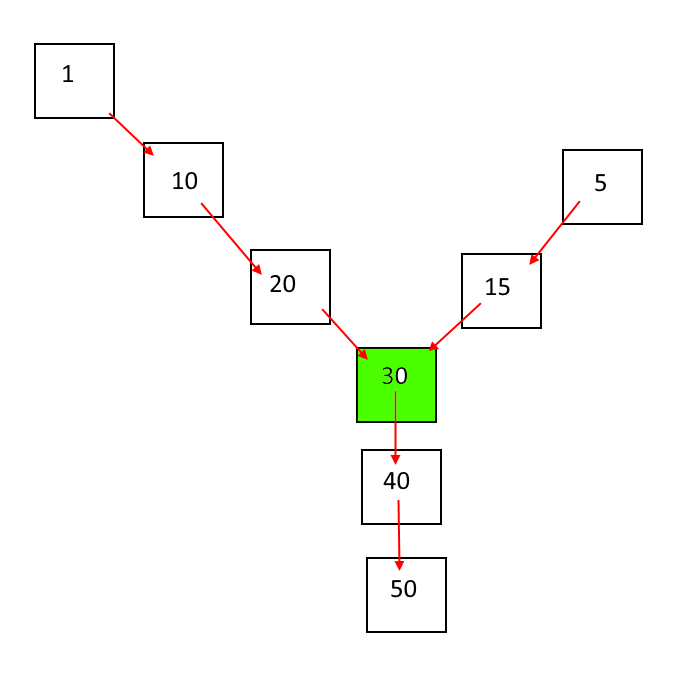
**Objective:** Given Two linked list, check whether both list intersect each other, if yes then find the starting node of the intersection.

Intersection point means end of one linked list is linked with some node in another linked list and it forms a Y shape.

**Input:** Two Linked List

**Output: Intersection Node or point**

**Example:**

****

**Approach:**

* Find the length of both the linked lists say : a\_len and b\_len
* Find the lenDiff = (a\_len ~ b\_len)
* Traverse the longer linked list by lenDiff
* Now traverse both the lists at the same time
* Check whether nodes are same, if yes then we have found the intersection point
* If we reach the end of the link lists then there is no intersection point.

**Trick Solution:**

* Take one linked list and join it both ends.
* Nor for the second Linked List, the problem is reduced to "Find a loop in a linked list and find the starting point of the linked list. So see inplementation, Click here.
* Starting point will be our intersection point.

**Complete Code:**

**package** findIntersectionOfLinkedLists;

//find the length of both the linked lists say : a\_len and b\_len

//find the lenDiff = (a\_len ~ b\_len)

//traverse the longer linked list by lenDiff

//Now traverse both the lists at the same time

//check whether nodes are same, if yes then we have found the intersection point

//if we reach the end of the link lists then there is no intersection point.

**public** **class** FindIntersectionOfLinkedLists {

**public** LinkedListIntersection a;

**public** LinkedListIntersection b;

**public** **void** createLists(){

a = **new** LinkedListIntersection();

a.addAtEnd(1);

a.addAtEnd(10);

a.addAtEnd(20);

Node tmp = a.addAtEnd(30);

a.addAtEnd(40);

a.addAtEnd(50);

a.addAtEnd(60);

System.*out*.print("List A : ");

a.display();

b = **new** LinkedListIntersection();

b.addAtEnd(5);

b.addAtEnd(15);

b.createIntersection(a,tmp);

System.*out*.print("List B : ");

b.display();

}

**public** **void** findIntersectionByLength(){

**int** a\_len=0;

**int** b\_len=0;

**int** lenDiff=0;

**boolean** intsctFound = **false**;

Node an = a.head;

Node bn = b.head;

**while**(an!=**null**){

an=an.next;

a\_len++;

}

**while**(bn!=**null**){

bn=bn.next;

b\_len++;

}

an = a.head;

bn = b.head;

**if**(a\_len>b\_len){

lenDiff = a\_len-b\_len;

// System.out.print("length diff " +lenDiff );

**while**(lenDiff!=0){

an = an.next;

lenDiff--;

}

}**else**{

lenDiff = b\_len-a\_len;

**while**(lenDiff!=0){

bn = bn.next;

lenDiff--;

}

}

**while**(an!=**null** && bn!=**null**){

//System.out.print(an.data + " " + bn.data);

**if**(an==bn) {

System.*out*.print("Intersection found at " + an.data);

intsctFound = **true**;

**break**;

}

**else**{

an = an.next;

bn = bn.next;

}

}

**if**(intsctFound!=**true**){

System.*out*.print("Intersection Not Found");

}

}

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

FindIntersectionOfLinkedLists i = **new** FindIntersectionOfLinkedLists();

i.createLists();

i.findIntersectionByLength();

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListIntersection{

**public** Node head;

**public** LinkedListIntersection(){

head=**null**;

}

**public** Node addAtEnd(**int** data){

Node n = **new** Node(data);

**if** (head==**null**){

n.next = head;

head = n;

}

**else**{

Node currNode = head;

**while**(currNode.next!=**null**){

//System.out.print("---->" + currNode.data);

currNode = currNode.next;

}

currNode.next = n;

}

**return** n;

}

**public** **void** createIntersection(LinkedListIntersection a, Node nd){

Node hd = a.head; // this is the list to whcih another list will intersect, in our example its list a

**while**(hd!=nd){

hd = hd.next;

}

Node currNode = head;// this is for the list which will connect, in our example its list b

**while**(currNode.next!=**null**){

currNode = currNode.next;

}

currNode.next = hd; ;

}

**public** **void** display(){

System.*out*.println("");

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

System.*out*.println("");

}

}

**Output:**

List A :

->1->10->20->30->40->50->60

List B :

->5->15->30->40->50->60

Intersection found at 30

**Algorithms - Swap Every Kth Node in a LinkedList**

**Objective:** Given a linked list, swap every kth node in that. If at the end of the list remaining nodes are less than k, leave them untouched.

**Input:** A linked list, A number k.

**Example:**

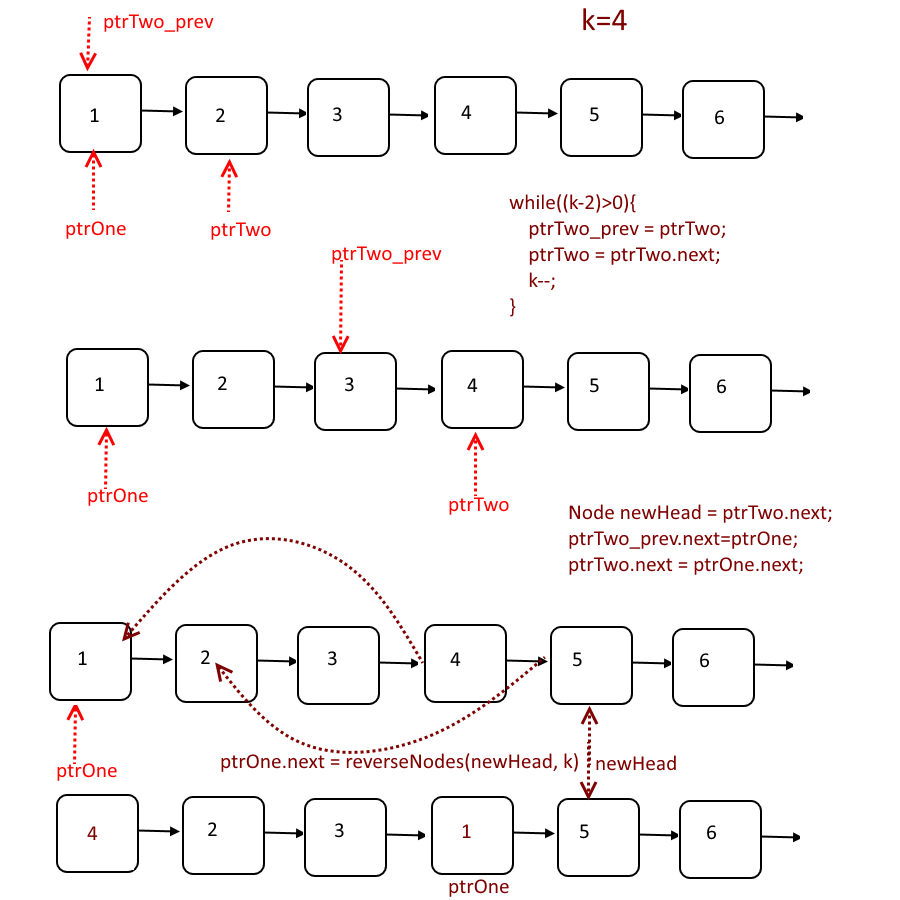
Input : ->1->2->3->4->5->6->7->8->9->10 , K = 4

Output: ->4->2->3->1->8->6->7->5->9->10

**Approach**:

* Take 3 Pointers, ptrOne, ptrTwo and ptrTwo\_prev.
* ptrOne and ptrTwo\_prev points at head node.
* ptrTwo points at next node of ptrTwo\_prev.
* Move the ptrTwo and ptrTwo\_prev k-2 times, since we need one pointer each at both ends for swapping so move pointers only k-2 times.
* Create another pointer , NewHead and point it to ptrTwo.next.
* Now we have ptrOne at head and ptrTwo at kth position, swap them with the help of ptrTwo\_prev.
* This function will returns the head.
* Now make a recursive call with newHead.

**ptrOne.next = reverseNodes(newHead, k);**

****

**Complete Code:**

**package** swapEverykthNodes;

**public** **class** SwapEveryKthNode {

**public** **static** **void** main (String[] args) **throws** java.lang.Exception

{

LinkedListT a = **new** LinkedListT();

a.addAtBegin(10);

a.addAtBegin(9);

a.addAtBegin(8);

a.addAtBegin(7);

a.addAtBegin(6);

a.addAtBegin(5);

a.addAtBegin(4);

a.addAtBegin(3);

a.addAtBegin(2);

a.addAtBegin(1);

System.*out*.print("Original Link List 1 : ");

a.display(a.head);

**int** k = 4;

Node n = a.reverseNodes(a.head, k);

System.*out*.println("\n Swap Every " + k + "th Node : ");

a.display(n);

}

}

**class** Node{

**public** **int** data;

**public** Node next;

**public** Node(**int** data){

**this**.data = data;

**this**.next = **null**;

}

}

**class** LinkedListT{

**public** Node head;

**public** LinkedListT(){

head=**null**;

}

**public** Node reverseNodes(Node head, **int** k){

**int** x =k;

Node ptrOne = head;

Node ptrTwo\_prev = head;

Node ptrTwo = **null**;

**if**(k<2)**return** head;

**if**(ptrOne!=**null**){

ptrTwo = head.next;

}**else** **return** **null**;

**while**((x-2)>0){

**if**(ptrTwo!=**null**){

ptrTwo\_prev = ptrTwo;

ptrTwo = ptrTwo.next;

x--;

}**else**{

**return** head;

}

}

Node newHead = ptrTwo.next;

ptrTwo\_prev.next=ptrOne;

ptrTwo.next = ptrOne.next;

ptrOne.next = reverseNodes(newHead, k);

**return** ptrTwo;

}

**public** **void** addAtBegin(**int** data){

Node n = **new** Node(data);

n.next = head;

head = n;

}

**public** **void** display(Node head){

Node currNode = head;

**while**(currNode!=**null**){

System.*out*.print("->" + currNode.data);

currNode=currNode.next;

}

}

}

**Output:**

Original Link List 1 : ->1->2->3->4->5->6->7->8->9->10

Swap Every 4th Node :

->4->2->3->1->8->6->7->5->9->10

**Algorithms - Find Whether Two Strings are Permutation of each other**

**Objective:** Given Two Strings, check whether one string is permutation of other

**Input:** Two Linked list.

**Output:** True or false based on whether strings are permutation of other or not.

**Example:**

**"sumit" and "tiums" are permutations of each other.**

**"abcd" and bdea" are not permutations of each other.**

**Approach:**

**Method 1: Time Complexity - O(nlgn)**

Sort both the strings and compare it.

**Method 2 : Using Hash Table - Time Complexity - O(n)**

* Check if both Strings are having the same length, if not , return false.
* Create a Hash Table, make character as key and its count as value
* Navigate the string one taking each character at a time
* check if that character already existing in hash table, if yes then increase its count by 1 and if it doesn't exist insert into hash table with the count as 1.
* Now navigate the second string taking each character at a time
* check if that character existing in hash table, if yes then decrease its count by 1 and if it doesn't exist then return false.
* At the end navigate through hash table and check if all the keys has 0 count against it if yes then return true else return false.

Complete Code:

**package** newInterviewQuestions;

**import** java.util.Hashtable;

**import** java.util.Iterator;

**import** java.util.Set;

**public** **class** PermutationStrings {

**public** **boolean** isPermutation(String s1, String s2){

**if**(s1.length()!=s2.length()){

**return** **false**;

}

Hashtable<Character , Integer> ht = **new** Hashtable<>();

**for**(**int** i=0;i<s1.length();i++){

**char** c = s1.charAt(i);

**if**(ht.containsKey(c)){

**int** val = ht.get(c) +1;

ht.put(c, val);

}**else**{

ht.put(c, 1);

}

}

**for**(**int** i=0;i<s2.length();i++){

**char** c = s2.charAt(i);

**if**(ht.containsKey(c)){

**int** val = ht.get(c);

**if**(val==0){

**return** **false**;

}

val--;

ht.put(c, val);

}**else**{

**return** **false**;

}

}

Set<Character> keys = ht.keySet();

**for**(Character k :keys ){

**if**(ht.get(k)!=0){

**return** **false**;

}

}

**return** **true**;

}

**public** **static** **void** main(String args[]){

String s1 = "sumit";

String s2 = "mtisu";

PermutationStrings p = **new** PermutationStrings();

System.*out*.println(s1 +" and "+ s2 + " are permutation of each other? " + p.isPermutation(s1, s2));

s1 = "xyzab";

s2 = "bayzxx";

System.*out*.println(s1 +" and "+ s2 + " are permutation of each other? " + p.isPermutation(s1, s2));

}

}

Output:

sumit and mtisu are permutation of each other? true

xyzab and bayzxx are permutation of each other? false

**Algorithms - Replace all spaces in a String with '%20'**

**Objective:** In a given string , **replace all spaces in a String with '%20'. You can consider that string has enough space at the end of the string to hold the extra characters.**

**Input:** A String and true length of a string

**Output:** Updated string in which each space is replaced by the '%20'

**Example:**

Input String : I am Sumit Jain

Output String : I%20am%20Sumit%20Jain

**Approach:**

* Count the total spaces in a string in one iteration, say the count is spaceCount
* Calculate the new length of a string by newLength = length + 2\*spaceCount; (we need two more places for each space since %20 has 3 characters, one character will occupy the blank space and for rest two we need extra space)
* Do another iteration in reverse direction
* If you encounter the space, for next 3 spaces put %,2,0.
* If you encounter the character, copy it

Complete Code:

**package** newInterviewQuestions;

**public** **class** ReplaceAllSpaces {

**public** **void** replace(String s1, **int** length){

**char** [] chars = s1.toCharArray();

**int** spaceCount =0;

**for**(**int** i=0;i<length;i++){

**if**(chars[i]==' '){

spaceCount++;

}

}

**int** newLength = length + 2\*spaceCount;

**for**(**int** i=length-1;i>=0;i--){

**if**(chars[i]==' '){

chars[newLength-1]='0';

chars[newLength-2]='2';

chars[newLength-3]='%';

newLength = newLength-3;

}**else**{

chars[newLength-1]=chars[i];

newLength = newLength-1;

}

}

s1 = String.*valueOf*(chars);

System.*out*.println("Output String : " + s1);

}

**public** **static** **void** main(String args[]){

String s1 = "I am Sumit Jain ";

**int** trueLength = 15;

System.*out*.println("Input String : " + s1);

ReplaceAllSpaces r = **new** ReplaceAllSpaces();

r.replace(s1, trueLength);

}

}

Output:

Input String : I am Sumit Jain

Output String : I%20am%20Sumit%20Jain

**Algorithms - String Compression using count of repeated characters.**

**Objective:** Compress the given string by using the count of repeated characters and if new compressed string length is not smaller than the original string then return the original string.

**Example:**

Input String : ssssuuuummmmmmiiiittttttttttttt

Compressed String : s4u4m6i4t13

Input String : Jaain

Compressed String : Jaain (Since compressed string is length is greater than original string)

**Input:** A String

**Output: either** A compressed string or original string whichever us smaller.

**Approach:**

* Create a StringBuffer sb, int count
* Navigate the string taking each character at a time.
* If you find the same characters increase the count.
* if not then append the character and its count to the string buffer sb.
* reset the count value.
* Compare the length of compressed String and original and whichever is smaller return that string.

Complete Code:

**package** newInterviewQuestions;

**public** **class** StringCompression {

**public** String compression(String s1){

StringBuffer sb = **new** StringBuffer();

**int** count =1;

**char** prev = s1.charAt(0);

**for**(**int** i=1;i<s1.length();i++){

**char** curr =s1.charAt(i);

**if**(prev==curr){

count++;

}**else**{

sb.append(prev);

sb.append(count);

prev = curr;

count=1;

}

}

sb.append(prev);

sb.append(count);

**if**(s1.length()<sb.length()){

**return** s1;

}**else**{

**return** sb.toString();

}

}

**public** **static** **void** main(String args[]){

String s1 = "ssssuuuummmmmmiiiittttttttttttt";

StringCompression sc = **new** StringCompression();

System.*out*.println("Compression of " + s1 + " is : " +sc.compression(s1));

s1 = "Jaain";

System.*out*.println("Compression of " + s1 + " is : " +sc.compression(s1));

}

}

Output:

Compression of ssssuuuummmmmmiiiittttttttttttt is : s4u4m6i4t13

Compression of Jaain is : Jaain

**Algorithms - Check if one string is Rotation of another string**

**Objective:** **Check if one string is Rotation of another string**.

**Example:**

Input Strings : 'sumitjain' and 'tjainsumi'

Output : true

Input String : 'Jaain' and 'ainJ'

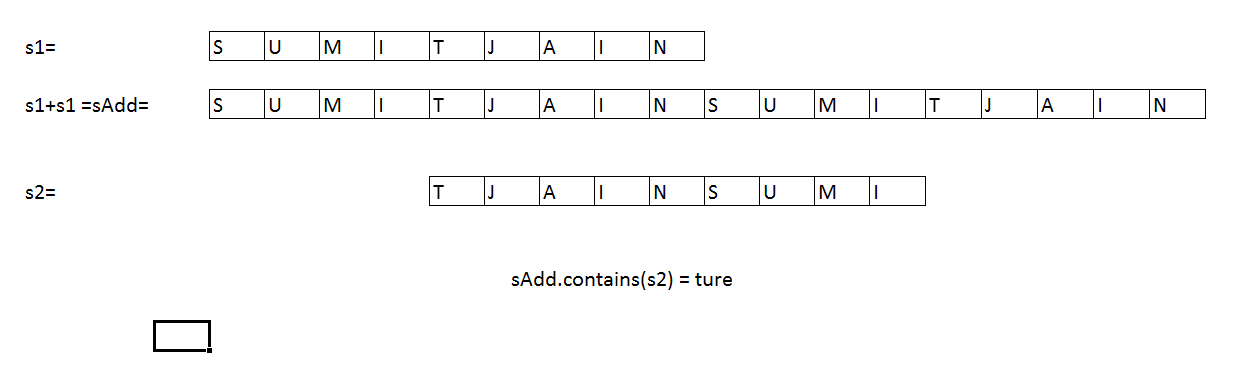
Output: false

**Input:** Two Strings

**Output: True or false based on whether strings are rotation of each other.**

**Approach:**

* Make a new String by appending the appending the first string with itself
* Check if second string is sub string of new String



Complete Code:

**package** newInterviewQuestions;

**public** **class** RotatedArray {

**public** **boolean** isRotated(String s1, String s2){

**if**(s1.length()!=s2.length()){

**return** **false**;

}

String sAdd = s1 + s1;

**if**(sAdd.contains(s2)){

**return** **true**;

}**else**{

**return** **false**;

}

}

**public** **static** **void** main(String arg[]){

String s1 = "sumitjain";

String s2 = "tjainsumi";

RotatedArray r = **new** RotatedArray();

System.*out*.println("Is '" + s1 + "' and '" + s2 + "' are rotated?? : " + r.isRotated(s1, s2));

}

}

Output:

Is 'sumitjain' and 'tjainsumi' are rotated?? : true

**Algorithms - Remove Duplicates from an unsorted Linked list.**

**Objective:** **Write a program to remove the duplicates from an unsorted linked list**

**Example:**

Input Linked List : 1->2->2->4->3->3->2

Output : 1->2->4->3->2

**Input:** An unsorted linked list

**Output: Linked list with no duplicates.**

**Approach:**

* Create a Hash Table
* Take two pointers, prevNode and CurrNode.
* PrevNode will point to the head of the linked list and currNode will point to the head.next.
* Now navigate through the linked list.
* Check every node data is present in the HashTable.
* if yes then delete that node using prevNode and currNode.
* If No, then insert that node data into the linked list
* Return the head of the list

Time Complexity : O(n)

Space Complexity : O(n)

**Follow Up**: If suppose addition buffer is not allowed then we have option but to check every node data against every other node data and if find duplicates, delete that node.

Time Complexity : O(n^2)

Complete Code for the Hash Table method:

**package** RemoveDuplicatesFromLL;

**import** java.util.Hashtable;

**public** **class** RemoveDuplicates {

**public** Node removeDup(Node head){

Hashtable<Integer, Integer> ht = **new** Hashtable<>();

**if**(head==**null**){

**return** **null**;

}

Node currNode = head.next;

Node prevNode = head;

**int** count =0;

**while**(currNode!=**null**){

**int** data = currNode.data;

**if**(ht.contains(data)){

prevNode.next = currNode.next;

currNode = currNode.next;

}**else**{

ht.put(count, data);

count++;

prevNode = currNode;

currNode = currNode.next;

}

} **return** head;

}

**public** **void** display(Node head){

Node n=head;

**while**(n!=**null**){

System.*out*.print("->" + n.data);

n=n.next;

}

}

**public** **static** **void** main(String args[]){

Node n = **new** Node(1);

n.next = **new** Node(2);

n.next.next = **new** Node(2);

n.next.next.next = **new** Node(3);

n.next.next.next.next = **new** Node(4);

n.next.next.next.next.next = **new** Node(4);

n.next.next.next.next.next.next = **new** Node(2);

System.*out*.print("Original List : ");

RemoveDuplicates rm = **new** RemoveDuplicates();

rm.display(n);

System.*out*.print("\n Updated List: ");

Node x =rm.removeDup(n);

rm.display(x);

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

Output:

Original List : ->1->2->2->3->4->4->2

Updated List: ->1->2->3->4

**Algorithms - To find the kth to Last Element of a Singly Linked List**

**Objective:** **Write a program to find the kth to Last Element of a Singly Linked List**

**Example:**

Original List : ->1->2->8->3->7->0->4

Output : 3rd Element from the end is : 7

**Input:** An unsorted linked list and integer k

**Output: The kth to Last Element of a Singly Linked List**

**Approach:**

**Recursive Approach:**

* Recurse through the Linked list
* When we reach to the end of the list, base case will return 0
* Now with each passing back call through stack, add 1 and return.
* When count hits k, print the value.

**Iterative Approach:**

* Take two pointers approach
* Move first pointer by k
* now move both the pointers and when the first pointer reaches the end of the list the second pointer will be at the kth node from the end.
* Return the kth node data.''

**Complete Code:**

**package** KthToLastElementofLL;

**public** **class** KthToLastElementofLL {

**public** **int** kthByRecursion(Node head, **int** k){

**if**(head==**null**){

**return** 0;

}

**int** i = kthByRecursion(head.next, k)+1;

**if**(i==k){

System.*out*.println(head.data);

}

**return** i;

}

**public** **int** kthByIteration(Node head, **int** k){

**if**(head==**null**){

**return** 0;

}

Node curr = head;

**while**(k>0){

curr=curr.next;

k--;

}

Node sec = head;

**while**(curr!=**null**){

curr = curr.next;

sec = sec.next;

}

**int** i = sec.data;

**return** i;

}

**public** **void** display(Node head){

Node n=head;

**while**(n!=**null**){

System.*out*.print("->" + n.data);

n=n.next;

}

}

**public** **static** **void** main(String args[]){

Node n = **new** Node(1);

n.next = **new** Node(2);

n.next.next = **new** Node(8);

n.next.next.next = **new** Node(3);

n.next.next.next.next = **new** Node(7);

n.next.next.next.next.next = **new** Node(0);

n.next.next.next.next.next.next = **new** Node(4);

System.*out*.print("Original List : ");

KthToLastElementofLL rm = **new** KthToLastElementofLL();

rm.display(n);

System.*out*.print("\n Recursion::3rd Element from the end is : ");

rm.kthByRecursion(n, 3);

System.*out*.print("\n Iteration::5th Element from the end is : " + rm.kthByIteration(n, 5));

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

Original List : ->1->2->8->3->7->0->4

Recursion::3rd Element from the end is : 7

Iteration::5th Element from the end is : 8

**Algorithms - Delete a Node in the Middle of a linked list, Given only access to that Node**

**Objective:** **Write a program to Delete a Node in the Middle of a linked list, Given only access to that Node**

**Example:**

Original List : ->1->2->8->3->7->0->4

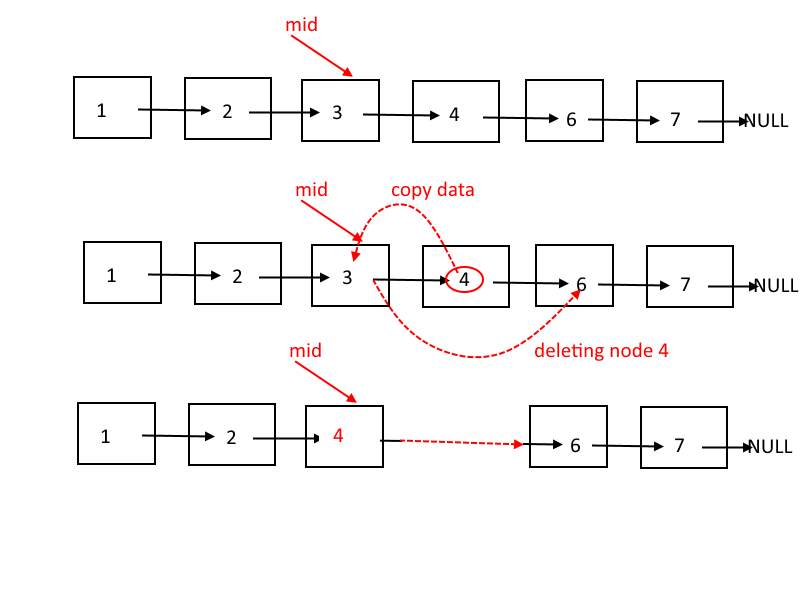
After Deleting the mid node (say 7) : ->1->2->8->3->0->4

**Input:** A Linked List and access to the node which needs to be deleted

**Output: Linked list with deleted node**

**Approach:**

* Approach is tricky and simple
* Copy the value of next node to the node which you want to delete
* Delete the next node



**Complete Code:**

**package** deleteMiddleNodeInLL;

**public** **class** deleteMiddleNodeInLL {

**public** **void** deleteMiddle(Node mid){

**if**(mid.next==**null**){

**return**; // we cant delete the node if it is the last node in the linked list

}

Node curr = mid;

curr.data = curr.next.data;

curr.next = curr.next.next;

}

**public** **void** display(Node head){

Node n=head;

**while**(n!=**null**){

System.*out*.print("->" + n.data);

n=n.next;

}

}

**public** **static** **void** main(String args[]){

Node n = **new** Node(1);

n.next = **new** Node(2);

n.next.next = **new** Node(8);

n.next.next.next = **new** Node(3);

Node mid = **new** Node(7);

n.next.next.next.next = mid;

n.next.next.next.next.next = **new** Node(0);

n.next.next.next.next.next.next = **new** Node(4);

System.*out*.print("Original List : ");

deleteMiddleNodeInLL rm = **new** deleteMiddleNodeInLL();

rm.display(n);

System.*out*.print("\n Aftter Deleting the mid node : ");

rm.deleteMiddle(mid);

rm.display(n);

}

}

**class** Node{

**int** data;

Node next;

**public** Node(**int** data){

**this**.data = data;

next = **null**;

}

}

Original List : ->1->2->8->3->7->0->4

Aftter Deleting the mid node (say 7) : ->1->2->8->3->0->4

**Algorithms - Reverse the binary representation of a number.**

**Objective:** **Write a program to Reverse the binary representation of a number**

**Example:**

Input : 30

Output : 15

**Explanation**:

binary representation of 30 is : 11110

reverse of binary representation : 01111

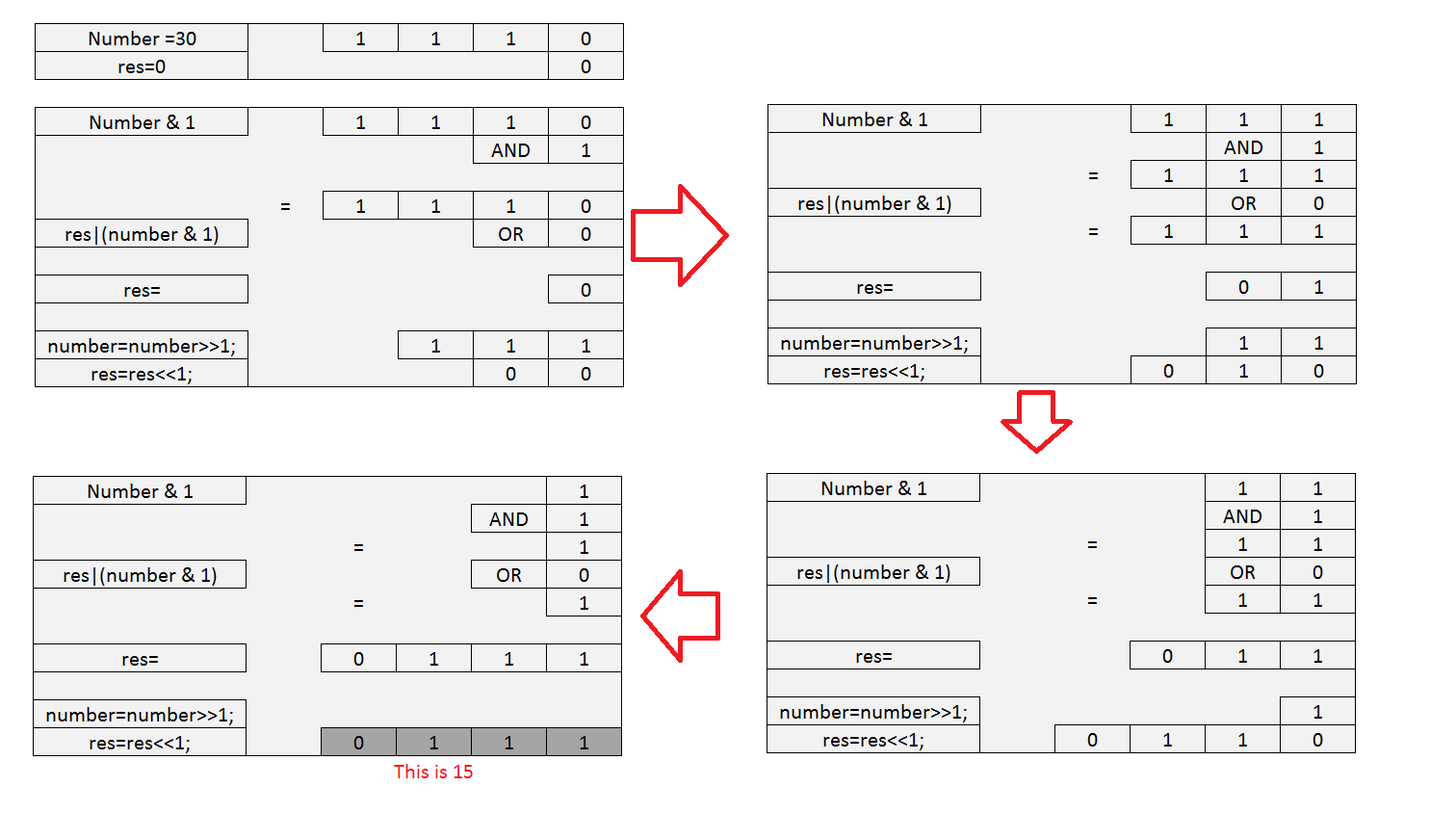
decimal of reversed binary representation is : 15

**Input:** A Number

**Output: Decimal of reversed binary representation of a number.**

**Approach:**

* Initialize int res =0
* Now from a number , take one bit at a time
* take AND of that bit with 1 and then OR with res and store it in res
* make right shift in number by 1
* make left shift in res by 1



**Complete Code:**

**package** newInterviewQuestions;

**public** **class** BinaryRotate {

**public** **int** rotateBinary(**int** number){

**int** res = 0;

**while**(number>0){

res=res<<1;

res = res|(number & 1);

number=number>>1;

}

**return** res;

}

**public** **static** **void** main(String args[]){

**int** x =30;

BinaryRotate b = **new** BinaryRotate();

System.*out*.println("Binary rotation of "+ x + " is : " + b.rotateBinary(x));

}

}

**Output:**

Binary rotation of 30 is : 15