16/6

DATA STRUCTURES

{3, 1, 3, 2, 4} - Find the duplicate value, the size of array is max_value + 1.

Brute Force
optimal
Best.

	Time	Space
	O(nlogn)	O(1)
	O(n)	O(n)
	O(n)	O(1)

i) First Approach (Sorting)

Sorting and checking the nearest element by iterating each

{3, 1, 3, 2, 4}

1, 2, 3, 3, 4

Arrays.sort()

-It will use Merge Sort Algorithm

ii) HashMap Approach (Space comp: O(n))

Storing in Hashmap
and returning value having count greater than 1

iii) Optimized Approach.

- No additional HashMap
- Instead hashing in the array itself.

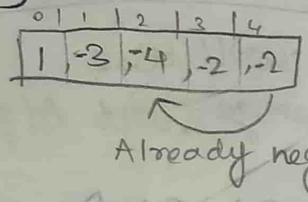
0	1	2	3	4
3	-1	-3	2	4

Negate the value of array element in its respective index.

```
for(int i=0; ... )
{ if(arr[i] < 0)
    { return true;
    }
  arr[arr[i]] = -1 * arr[arr[i]]
```

nums = { 1, 3, 4, 2, 2 } Dry Run ①

```
for(int i=0; i<nums.length; i++)
{
    int index = Math.abs(nums[i]); ②
    if(nums[index]<0) ③
    {
        SOPln("Yes there is a duplicate"); ④
    }
    else
    {
        nums[index] = -nums[index]; ⑤
    }
}
```



NEXT PERMUTATION (LC·31)

nums = [1, 2, 3]

Next biggest lexicographical combination.

① From right side, find the index of 1st decreasing (i)

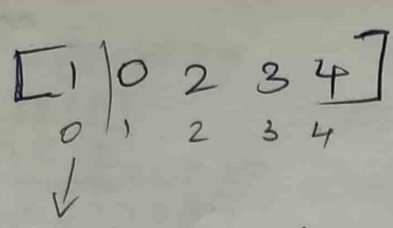
② From right to i, find the first greater element index (j)

③ swap(a[i], a[j])

④ Reverse from i+1.

1, 3, 5, 4, 2

MAX CHUNKS TO MAKE SORTED ARRAY. (LC- 769)



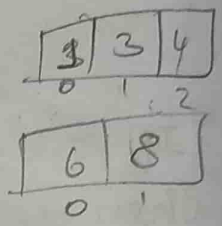
If you take this chunk alone then after sorting 1 will not be able to index 1. So here we cant chunk.

```
int maxSoFar
int chunks = 0;
for (int i=0; i< arr.length; i++)
{
    maxSoFar = Math.max(arr[i], maxSoFar);
    if (maxSoFar == i)
    {
      chunks++;
    }
}
return chunks;
}
```

Here we split chunk only when the maxsofar value eq to index because, maxSoFar value will be the last value of chunk after sorting, so have to split the chunk only when it reaches index

Take 2 array add the values & add in resultant array

```
int[] a1 = {1,3,4}
int[] a2 = {6,8};
```



// We have to handle carry

And also store the result sum in some structure like list.

```
List<Integer> li = new ArrayList<>();
int carry=0;
for(int i=a.length-1, j=b.length-1; i>=0||j>=0;
        i--, j--)
{
    int val = (i>=0?a[i]:0) +
              (j>=0?b[j]:0);
    int rem = val%10;
    carry = val/10;
    li.add(rem);
}
if(carry!=0)
    li.add(carry);
Collections.reverse(carry);
```

This part is important because at some iterations in the last iteration we get carry but we not proceed further iteration to add carry, so checking seperately and add!

(or)

add (carry > 0)

Condition in for loop

i>=0 || j>=0 || carry

LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS.

s = "abcabcbb"

Set [abc]

abcabcbb

18/6

TWO SUM - LC 1

[2, 7, 11, 15] target = 9

Using 2 pointers if the array is sorted

- Left & right pointer
- if sum of left + right is less than target increase left pointer, else right--
- when equal return index ie. left & right in an array
  - return new int[]{left, right}

else
  - return new int[2]; //empty array.

If not sorted then we can use method to sort along with index

```
class Pair
{ int value;
  int index;
  Pair(int value, int index)
  {
    this.value = value;
    this.index = index;
  }
}
class Solution {
    public int[] twoSum(int[] numbers,
        int len =                                    int targetSum
        Pairs[] pairs = new Pair[length];
```

```
for(int i=0; i<len; i++)
{   pairs[i]= new Pair(numbers[i], i);
}
Arrays.sort(pairs, (a,b) ->
        Integer.compare(a.value, b.value));
```

. . .

Code: yesamitsingh.hashnode.dev/tow-sum-leetcode

Three Sum (LC-15)

- First ~~i~~ iterate a for loop
- Then from i+1 posi to an len position perform 2 sum logic to check the target sum.

Max Sum SubArray (GFG)

Using SLIDING WINDOW.

[100, 200, 300, 400], k=2
OP: 700

```
int maxsubarray(int[] arr, int k)
{   int max=0
    int len = arr.length;
    for(int i=0; i<=len-k; i++)
    {
      int sum=0;
      int j=i;
      while(j<k+i)
       {
         sum+= arr[j++];
       }
      max = Math.max(max, sum)
    }}
```

The above approach is not efficient
Because at each index we sum loop k times
So use sliding window

[100, 200, 300, 400]

- First take window [100, 200]
- Remove first element and then add the next element

[100 (Rem) , 200 , 300 (Add) , 400]

[100 , 200 , 300 , 400]

k = 3

[1, 2, 3, 4, 5]

max = b

sum = max - arr[i-1] + arr[i+k-1]
= max

```
int sum=0
int max=0;
int len = arr.length;
for(int i=0; i<k; i++)
{   sum
    max += arr[i];
}max=sum;
for(int i=1; i<=len-k; i++)
{
   sum= max - arr[i-1] + arr[i+k-1];
   max= Math.max(sum, max);
}
return max;
```

Count occurances of Anagram

txt = aabaabaa pat = "aaba"

```
len = pat.length()
for(int i=0; i<=txt.len - pat.len; i++)
{
  String window = text.substring
                    (i, i+len);
  if(isAnagram(window, pat)){
       count++;
} }
```

aabaabaa
0 1 2 3 4 5 6 7

i = 1
win[0] = txt[i]
win[len] =

But, not efficient approach

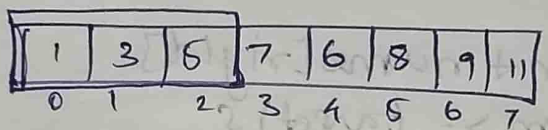
```
boolean isAnagram(int[] window, int[] pat)
{
 for(int i=0; i<window.length; i++)
 {
  if(window[i] != pat[i])
  {
   return false;
  }
 }
 return true;
}

int search(String pat, String txt){
    int count=0;
    int len=pat.length();
    int[] window=new int[26];
    int[] pat_window=new int[26];
    for(int i=0; i<len; i++)
    {
      window[txt.charAt(i)-'a']++;
      pat_window[pat.charAt(i)-'a']
                                  ++;
    }
    if(isAnagram(window,pat_window))
     {
       count++;
     }
    for(int i=len; i<txt.length(); i++)
    {
      window[txt.charAt(i-len)-'a']--;
      window[txt.charAt(i)-'a']++;
      if(isAnagram(window, pat_window))
       {
         count++;
       }
    }
    return count;
}
```

SLIDING WINDOW APPROACH

* First fill the initial window with a for loop. If condition is there to check then check.

* Then start to slide the window from the upcoming value.

Like if k=3 First window would



After this start next loop with i = k till i < len and check condition within iteration.

19/6/25

LONGEST SUBSTRING WITHOUT REPEATING CHARACTERS (LC 3)

s = "abcabcbb".

abcabcbb

abcabcbb

Set : a b c

MINIMUM SIZE SUBARRAY SUM (LC 209)

[2, 3, 1, 2, 4, 3]

target = 7.

2 5 6 8 12 16

2 3 1 2 4 3

len = 4

sum = 2 5 6 8 6 10 7

```
public int minSubArrayLen(int target,
                          int[] nums)
{
  int left = 0;
  int right = 0;
  int minLen = Integer.MAX_VALUE;
  int sum = 0;
  while (right < nums.length)
  {
    sum = sum + nums[right];
    while (sum >= target) {
      int size = right - left + 1;
      minLen = Math.min(size,
                        minLen);
      sum -= nums[left];
      left++;
    }
    right++;
  }
  return minLen == Integer.MAX_VALUE ?
                   0 : minLen;
```

Here we should use while loop only, because if use if(condition) to increase the left pointer, it would move only position, but only if we use while( ) then it would move the left pointer until the ~~value~~ sum >= target, then we start moving the right pointer.

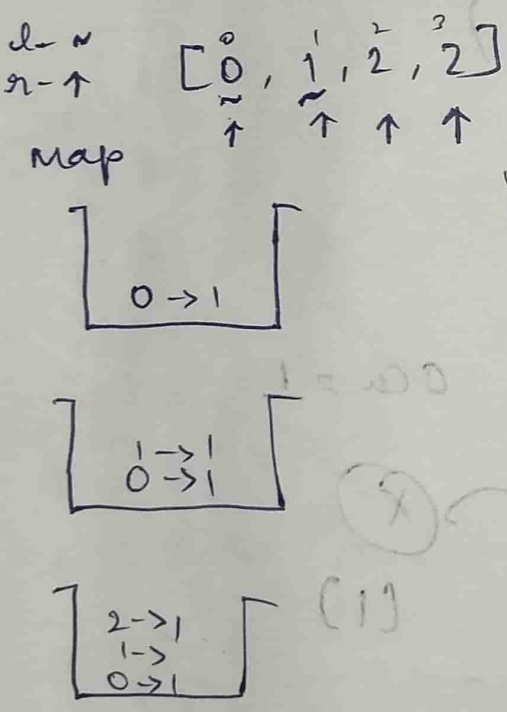
Fruits Into the Baskets

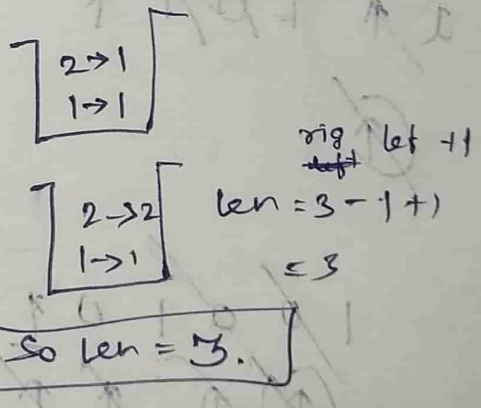
LC(904)

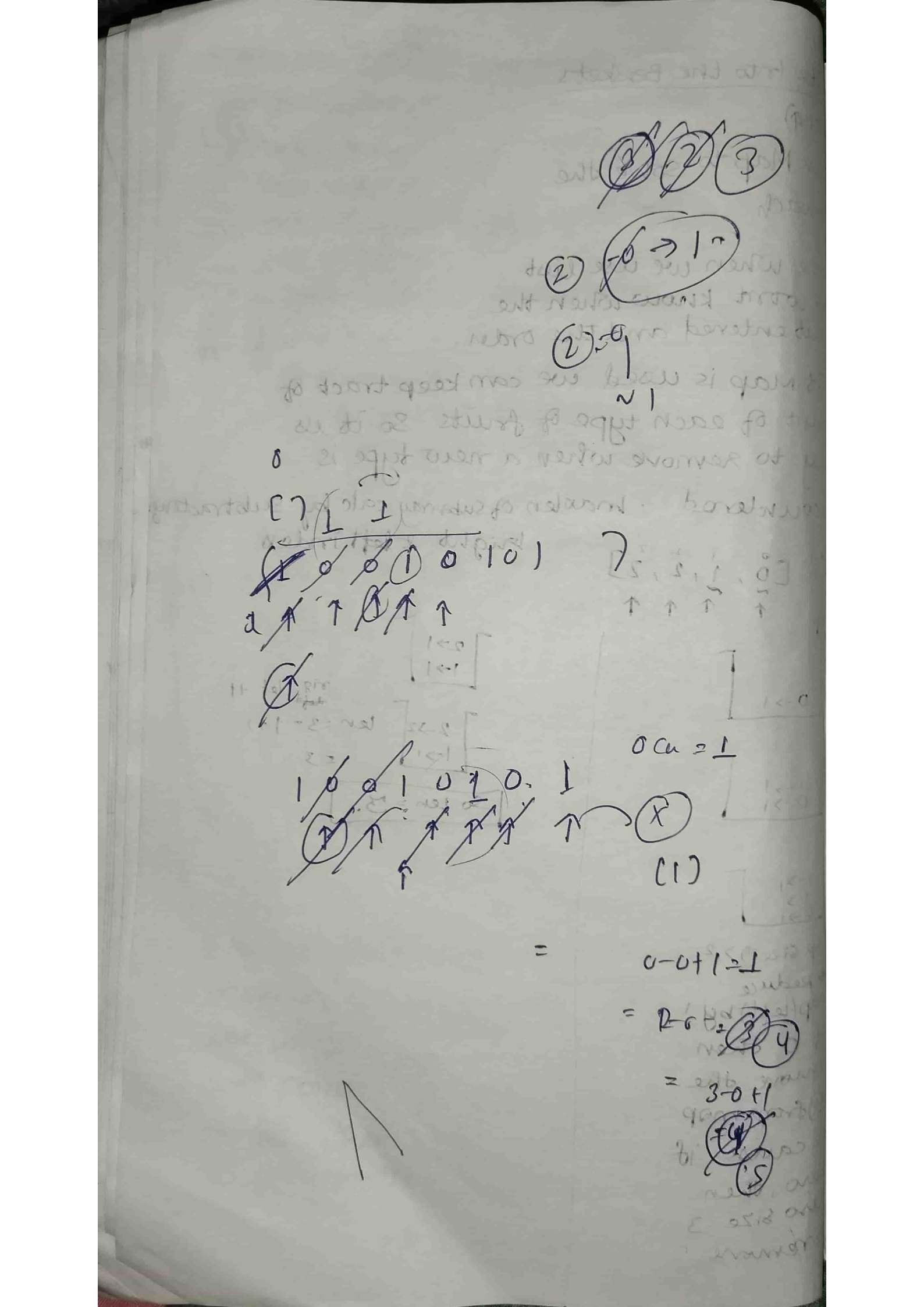
- Use Map to solve the approach

- Here when we use list we wont know when the fruit entered and the order.

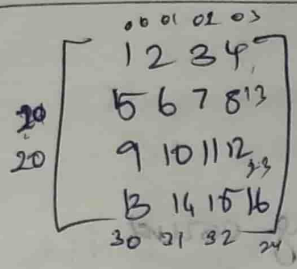
- If Map is used we can keep track of count of each type of fruits. So it is easy to remove when a new type is encountered. maxlen of subarray calc by subtracting right & left index



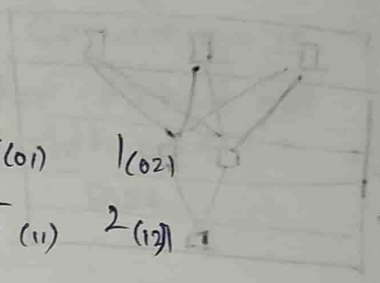




Matrix Boundary Traversal GFG.



Rotate Image LC - 48



1(00) 2(01) 3(02)
4(10) 5(11) 6(12)
7(20) 8(21) 9(22)

7(00) 4(01) 1(02)
8(10) 5(11) 2(12)
9(20) 6(21) 3(22)

i) Conv to transpose

1 4 7
2 5 8
3 6 9

ii) Reverse the rows of ~~linked list~~ matrix.

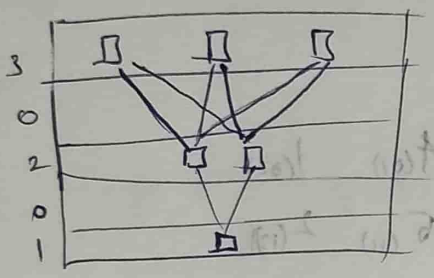
11 2 3
4 5 6
7 8 9

temp = mtrx

Number of Laser Beam.

LC (2125)

- Use prev & count
- prev to track the pre existing count.
- count to add the cumulative total.



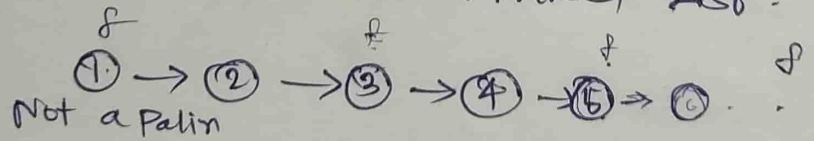
- We can't simply multiply 3 x 2 x 1.

Maintain prev to track previous non zero laser count, so that you can multiply with curr. Even if prev = 0 current can't have any laser beam with it.

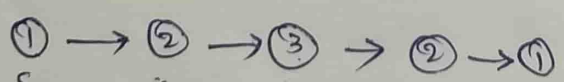
20/6/25

LC

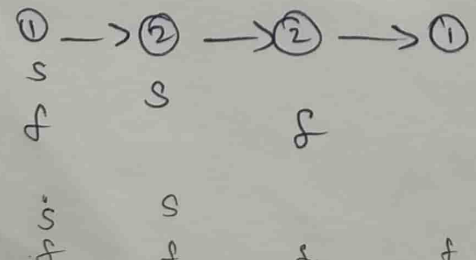
234. Palindrome Linked List.



Pal



Can use slow, fast pointer to find mid



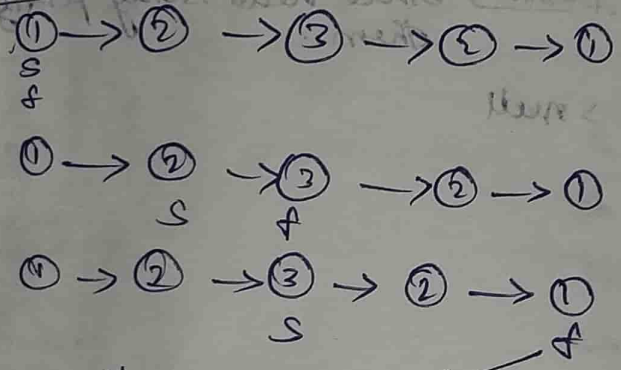
In this we have to use Floyd's Algorithm (ie. slow & fast pointer)
This would find length of the list in $n/2$ time complexity.

```
while (fast! = null & fast.next! = null)
```

↓ This is for linked list with even size

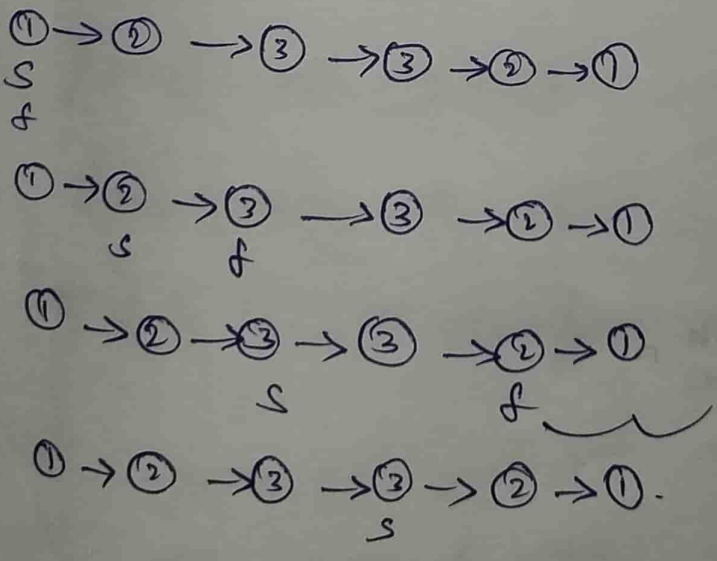
↓ Condition is for linked list with odd size.

Odd size



Here we check fast.next! = null and stop here.

Even size

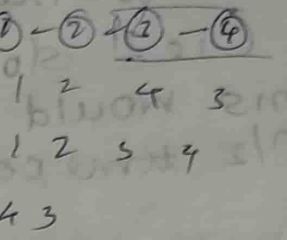


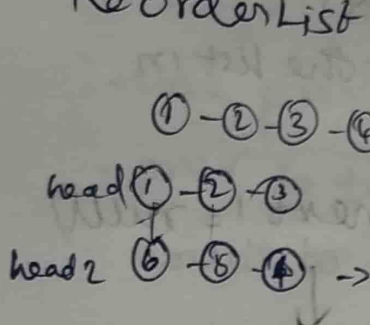
Here in even size we check fast! = null.

link: yesamitsingh.hashnode.dev/reorder-list

LC - 143.

Reorder List



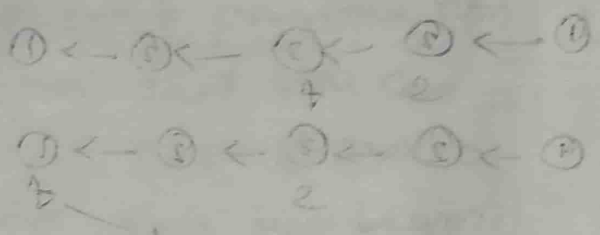


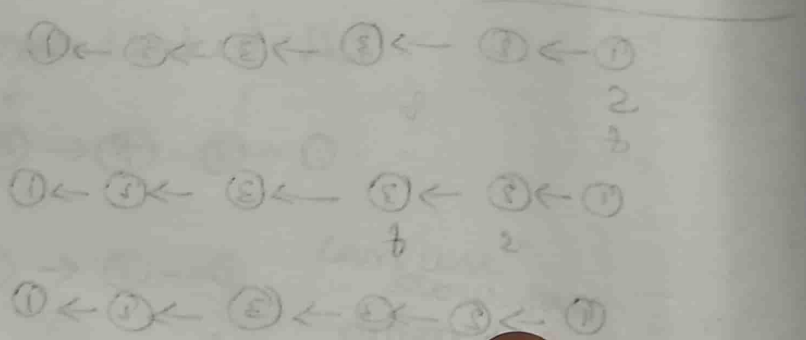
Even though we take till first half the remaining nodes are also available since head is only pointing them.

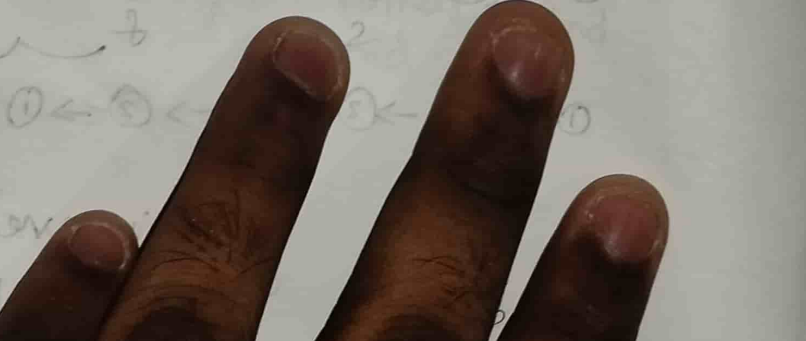
Head [1 → 2 → 3] → 4 → 5 → 6

slow is still

Head2 [6 → 5] → null







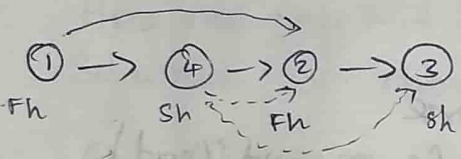
while reversing a linked list in reverse func use curr, prev, right pointer

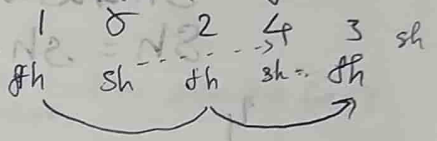
- and dont increement curr as curr = curr.next;
- instead increement by curr = prev;
because curr.next would be reversed to point prev.

UNFOLD OF LINKED LIST (Crafterhack)

Inp: ① → ④ → ② → ③

OP:



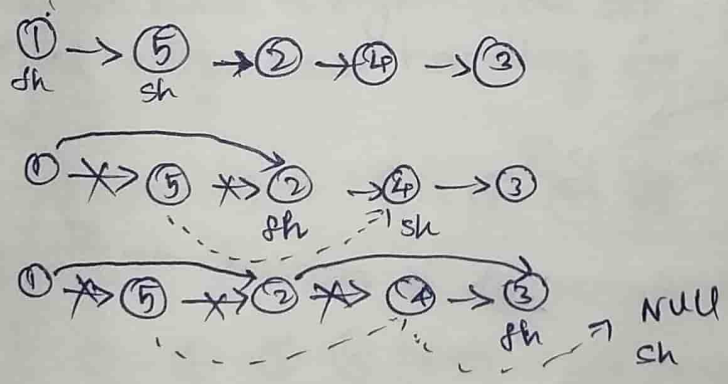


fh: 1 → 2 → 3
secondHead
sh: 5 → 4 ← sh

Fh = head
Sh = head.next

```
{ Fh.next = sh.next
  Fh = Fh.next
  sh.next = Fh.next
} Sh = sh.next;
```

Here while starting itself assign another pointer to sh (because that pointer is used to reverse) after reversing point fh → secondHead



fh : ① → ② → ③ (fh is here)

sh : ⑤ → ④ (sh is here.)

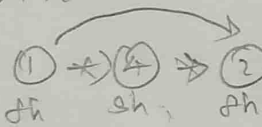
- So initially assign secondHead
- Now reverse (secondHead)
- Then fh → secondHead.

```
public ListNode unfoldLinkedList
                    (ListNode Head)
{
   ListNode fh = head;
   ListNode sh = head.next;
   ListNode secondHead = sh;
   while (sh != null && sh.next != null)
   {
      fh.next = sh.next;
      fh = fh.next;
      sh.next = fh.next;
      sh = sh.next;
   }
   ListNode head2 = reverse
                    (~~ListNode~~ secondHead);
   fh.next = head2;
   return head;
}
```

Here inside the while loop,

NOTE:

simply we should not move the pointer



Here sh from (4) is moved to (3)

by sh.next = fh.next;
sh = sh.next

we should not simply sh = fh.next;

If we did like that sh will be simply moved from (4) to (3), but we cant link (4) → (3). so we first link sh.next = fh.next, then we write sh = sh.next;

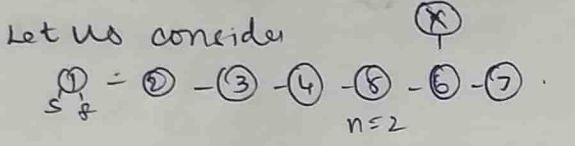
Remove Nth Node From End of List. LC.19

① → ② → ③ → ④ → ⑤.

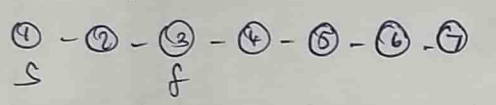
n = 2.

Then remove 2nd element from last.

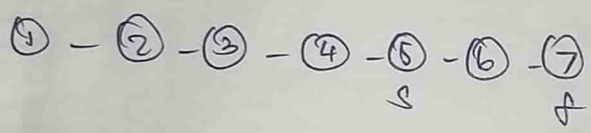
Let us consider




```
for(int j=0; j<n ; j++)
    fast++
```

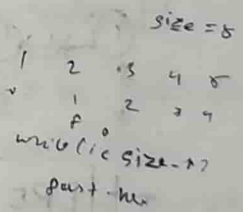


Here the gap will be the 'n' value betw slow & fast.
So when we move fast till it next becomes null



then fast will be at end and slow will be 2 (ie. n) before last element, Then the next element to slow will be the element to be deleted.

In this we have to ~~right~~ write an additional edge case handling for n == size



Because

if list size is 4 ① - ② - ③ - ④

and n = 4.

It means that fast passed all elements in linked list and pointing, which indicates that we want to remove head element so we return head.next.

LC-2

ADD TWO NUMBERS

l1 ① - ② - ③

l2 ④ - ⑧ - ⑥

res: ⑤ - ⑦ - ⑨

- Initialize a res node

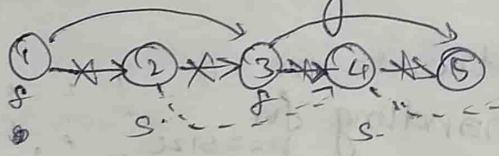
```
ListNode res = new ListNode();
```

- Then start from both pointers of l1 and l2 and move at same speed.
- Maintain a carry.

LC-328.

Odd Even Linked List

Same as that of Unfold Linked List done previously.

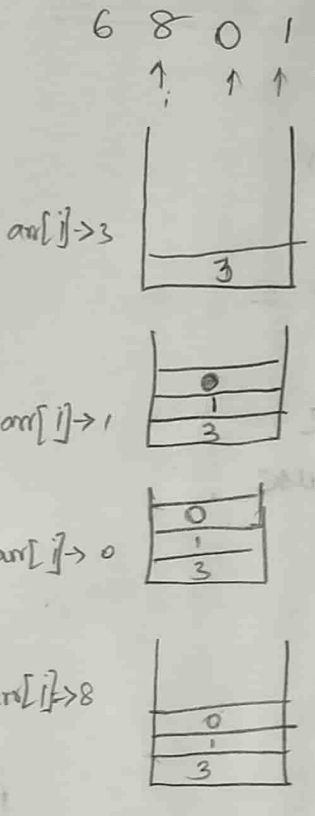


- HERE WE WONT REVERSE ALONE, BUT IN UNFOLD LINKEDLIST WE REVERSE THE HEAD2 LIST.

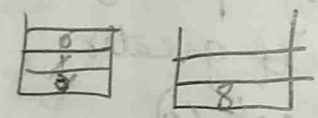
21/6/25

Next Greater Element (GFG)

- Using Stack
- Iterating array from the end.

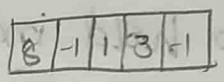


Stack.peek() < arr[i]
So remove until you find the greater element.



stack become empty
So adding -1 in res list.
Then push 8.

arr[] -> 6



max = -1
left < len
[1, 3, 2, 4]
[3]
6, 8, 0, 1, 3
max = -1
[8 8 0 1 3]
max = 8
max = arr[left+1]

Approach:

- Start from end of array.
- Check the stack if there is any next greater element present, if so add in result list by stack.pop() operation.
- If stack is empty then it means there is no next greater element So add (-1) in result.
- If stack.peek() is less than the curr element, then pop() until it finds greater element in stack. if it didn't find then
- After each step push the element in stack.

```
for(i = n-1; i>=0; i--)
    while(!stack.isEmpty() && stack.peek()<=arr[i])
            stack.pop();
    if(stack.isEmpty())
        res.add(0, -1)
    else
        res.add(0, stack.peek())
    stack.push(arr[i]);
```

Stock span Problem (GFG)

arr[] = [100, 80, 60, 70, 60, 75, 85]

op - [1, 1, 1, 2, 1, 4, 6]

From i^{th} index should go towards left, find the number of steps to go to find the first greater element.

Use a stack to hold indexes of each value

while moving through array check if arr[stack.peek()] > arr[i] if stack.peek() val is smaller remove (NOTE: We can remove that because it is not needed for upcoming element, since the curr element is greater)

Ex:

```
      i
      ↓
[ ... 60, 70 ... ]
```

If i is here remove 60 i.e index from stack. Because if any upcoming element is smaller to itself greater than 60, so it need not check 60)

So no damage on removing the index position.

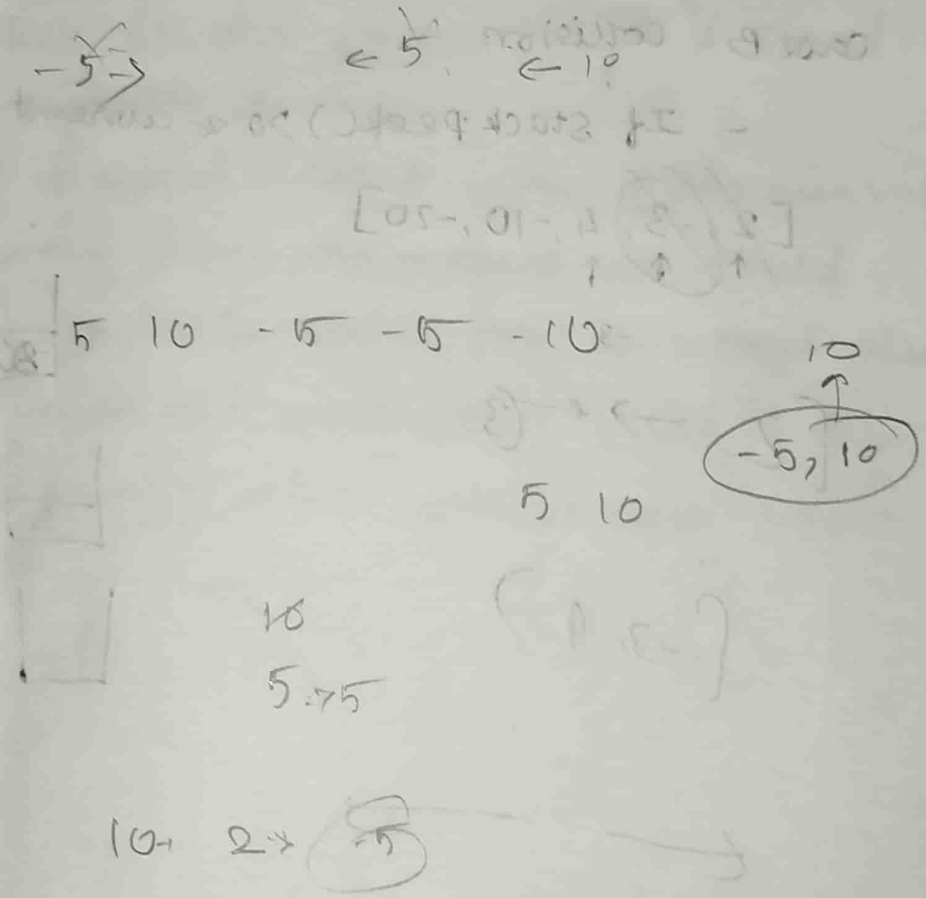
Both previous & this sum is Stack but that handles values, this one index.

Largest Rectangle in Histogram

LC-84

yeoamitsingh.hashnode.dev/histogram..

Asteroid Collision LC-735



[2, -3, 4, -10, -20].

- Take a stack
- Stack usually stores for moving right value. So when any left passing asteroid comes it checks the stack top and if any positive (right pass asteroid) , either of them will be popped
- After popping the unpopped asteroid is pushed inside the stack.
- If negative asteroid meets positive asteroid which is small, the positive popped and negative pushed

For each asteroid

Case A: No collision

- If stack empty.
- If current asteroid is moving right (>0)
- If top of stack is moving left (<0)

Case B: collision

- If stack.peek() >0 & current is <0.

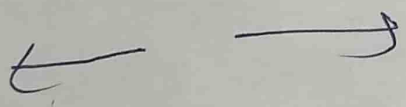
[2, -3, 4, -10, -20]

2 → 4 →

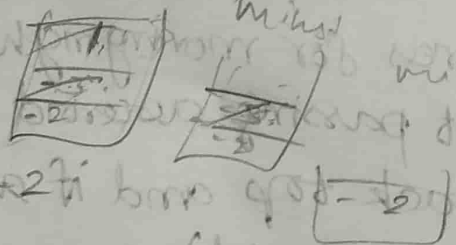
-3

[2 → ~~3~~

[-3, 4]



-2, -3, 1



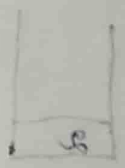
Minstack LC

Use two stacks.

One stack stores the values

Another one traces the minvalue keeps the history of changing min value. Because when we pop from the mainstack if the min value is popped then we lose the min count if stored in any separate variable.

But if stored in stack, even if the min value is removed from mainstack we would remove the same value from minstack also. Then the second most value that is the current peek() value will be min val in minstack.



22/6/24

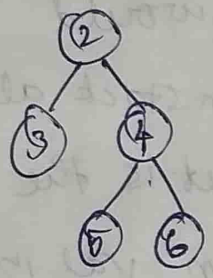
TREES

Preorder < Recursive / Iterative

Postorder < Rec / Iter

Inorder < Rec / Iter.

Preorder.

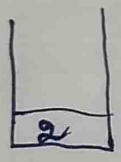


CODE

```
while(!stack.isEmpty())
    TreeNode curr = stack.pop();
    SOP(curr.val);
    if(curr.right != null)
        stack.push(curr.right);
    if(curr.left != null)
        stack.push(curr.left);
```

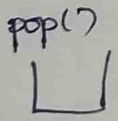
Here we first check curr.right node and then check curr.leftnode, because stack follows LIFO approach. So we pop the left val on top and then write.

1) at step

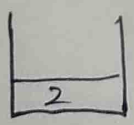


print 2

pop()

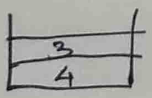


Iteration

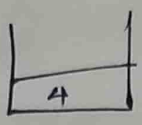


pop()

print 2



ii) curr = st.pop()



curr = 3

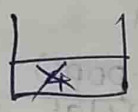
print 2 3

if (curr.right != null)

Nothing push.

iii) curr = st.pop()

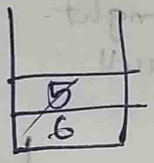
curr = 4



print 2 3 4

if right push

if left push

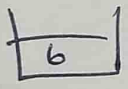


iv) curr = st.pop()

curr = 5

print 2 3 4 5

Nothing push



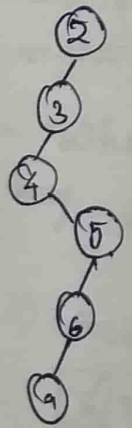
v) curr = st.pop()

curr = 6

print 2 3 4 5 6

End.

BINARY TREE INORDER TRAVERSAL LC-94



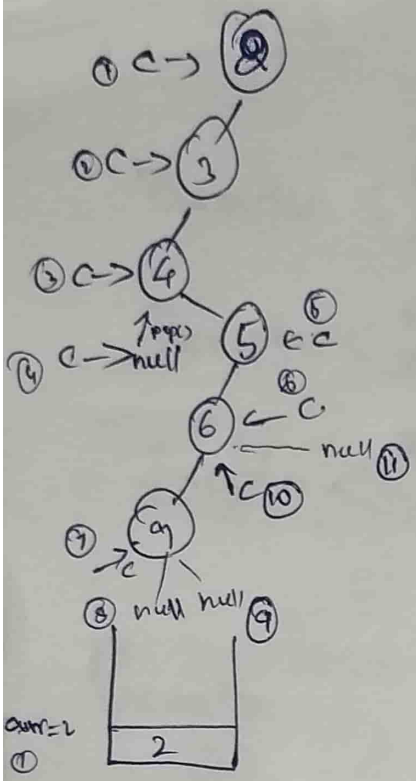
Move the curr ~~till~~ left side till you find the null ie. end pos, for each push in stack.

If reached curr.~~next~~ left = null

stop there and print.

Then check for right side and perform same task for each left movement push in stack

when stack is empty break.



curr=null

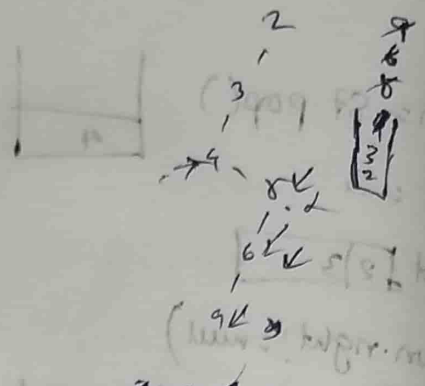
④ so pop() and move to right and traverse the same

print 4

⑤ curr=5

⑥ curr=6

⑦ curr=9



⑧ curr = null
so curr = st.pop()
then print 4 9

curr move to right
curr = null

⑨ curr = null

⑩ curr = st.pop()
curr = 6.

4 9 6

⑪ curr = st.pop()
curr = 5

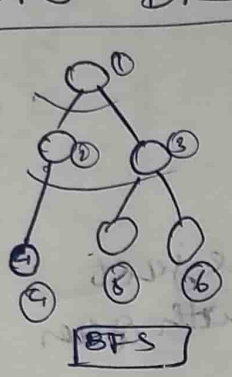
4 9 6 5

4 9 6 5 3

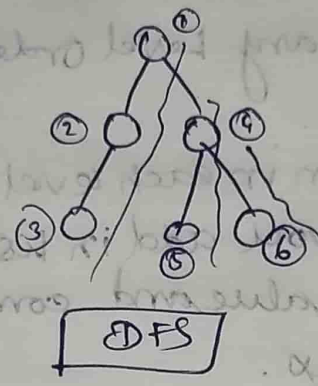
9 9 6 5 3 2

LEVEL ORDER TRAVERSAL

BFS DFS



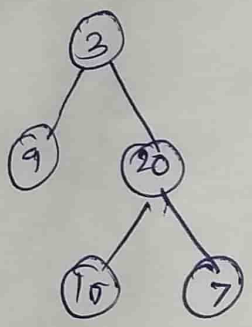
Level by level

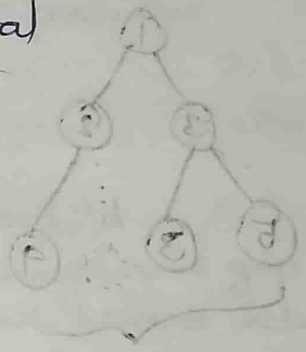


Till depth.

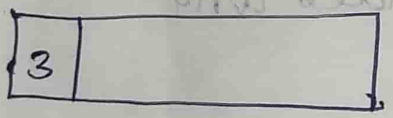
Binary Tree Level Order Traversal

LC - 102





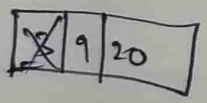
Maintain a queue to trace values.



~~while~~

- Take the size of ~~q~~ queue in 'size' variable.
- Iterate from i=0 to i<size times to push the left and right nodes into the queue.
- When adding left & right nodes of curr node ~~pop~~ remove() the node curr as well.

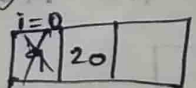
i) size = 1



list(3)

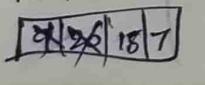
ii) size = 2

i=0



NO val.

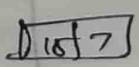
i=1



15, 7

list(9, 20)

iii)



list(15, 7)

MAX LEVEL SUM.

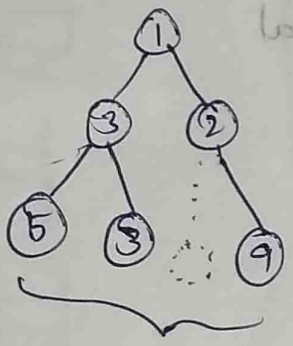
(1161)

-Same as Binary Level order Traversal

- Calculate sum in each level of tree

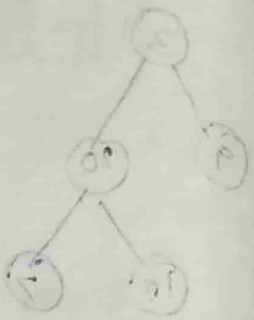
- In traversal you add in list, here just ~~comp~~ sum the value and compare with other values for max.

Maximum Width of Binary Tree. LC



4 nodes

So max width - 4



- Here we have to use a formula to find the length of left most index and right most index.

FORMULA:

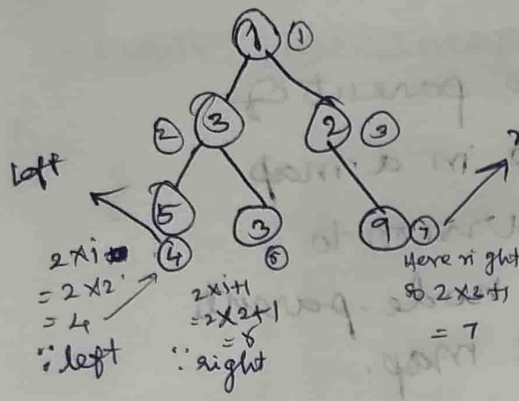
index for left - 2 x i

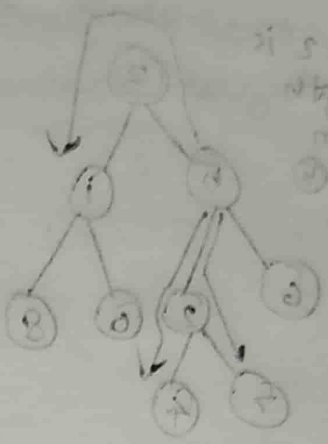
index for right - 2 x i + 1

Here i is the index of left & right's parent node.

So create a class Pair which stores the node along with its index.

```
class Pair{
    TreeNode node;
    int index
    Pair() {
        this.node
        this.ind
}}
```



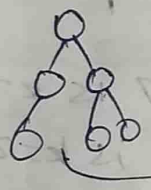


right - left + 1
= 7 - 4 + 1 = A

Maximum Width of Tree (GFG)

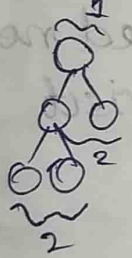
This is simple compared to prev one

Previous one is binary



- It can have one/two child
- So here there is gap
And we have to include that gap also in our width.

But in this sum it is complete binary tree.

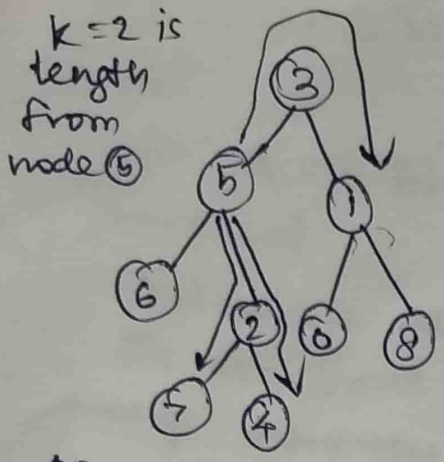


It is continous. So simply we can perform BFS and add the count of nodes in the level and return the max.

All Nodes Distance K in Binary Tree

LC - 863

Yesanitsingh.hashnode.dev/k-distance

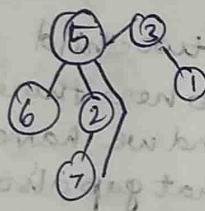


OP: 7, 4, 1

First store parent of each node in a map. Use recursion to store the node-parent values in map.

Then normally perform BFS to add the nodes at each level in the Queue in normal traversal you add only left and right. But in here, you have to add the parent of the node because treat it like a graph.

Ex:



Here 5 → 2 → 7 is k=2
and 5 → 3 → 1 is k=2
so we have to take ~~both~~ parent also.

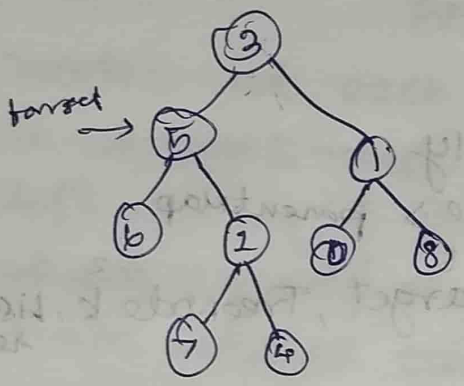
Use a set to track the visited nodes because in BFS we tend to visit same nodes again.

```
while(!q.isEmpty())
{
    if(k==0)
        return;

    k--;
}
```

Here k is the total level. we have to perform the operation at k levels. If 2, then we have to go till 2nd level from node 5 so k-- & if k=0

```
public List<Integer> distanceK(TreeNode root,
                               TreeNode target, int k)
{

}

Public
//Create the HashMap Globally.
 Map <TreeNode, TreeNode> parentMap
public void bfs( TreeNode target, TreeNode k, List result)
{
    Queue< Inte TreeNode> q = new LinkedList<>()
Set Stack< TreeNode> visited_stack = new Stack<>();
    queue.add (target)
visit_stack.add (target)
    while (!q.isEmpty())
    {
        if(K==0) break;
        int size = q.size();
        for(int i=0; i< size; i++)
        {
            TreeNode curr= q.remove());
            if(curr.left!=null && !visit_stack.cont
                                    (curr.left)
            {
                q.add(current.left);
            } visit_stack.add(current.left)
            if(curr.right!=null && !visit_stack.contains
                                    (curr.right)
            {
                q.add( )
            }       vis_stack.add()

            if(parentMap.contains(current) &&! visited.cont
                                    (par.get(current))
            {
                q.add(par.get(cur));
            } vis_stack.add(par.get(cur));
```

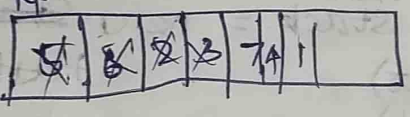
```
        k--;
      }
    }
    while( !q.isEmpty())
    {
      res.add(q.remove().val());
    }
```



5	3
6	5
2	5
7	2
4	2
1	3
0	1
8	1

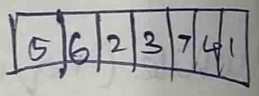
BFS

Queue

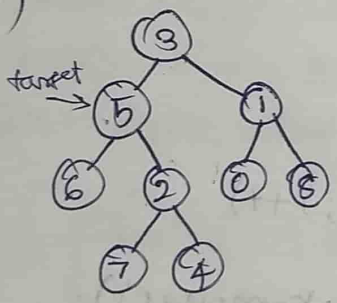


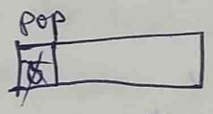
~~Stack~~

Set

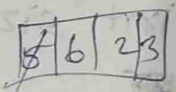


i) K = 2 (It means 1st level)





curr = 5

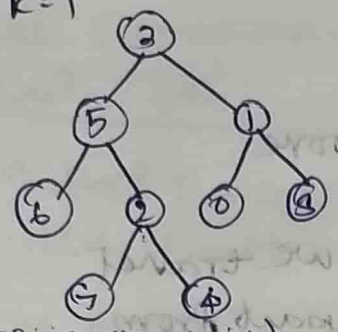


5. left = 6 Add in q/set

5. right = 2 q/set

5. parent = 3 (It is found from map) q/set

(ii) k=1



6	2	3

for (i=0; i<qsize=3; i++)

curr = 6

~~6~~	2	3

curr.left = null (Nothing add in q/set)
curr.right = null (Nothing add in q/set)
curr.parent = 5 (Already present in set, so nothing to add)

curr = 2

~~6~~	~~2~~	3		

curr.left = ~~null~~ 7 (Add in q/set)
curr.right = 4 (Add in q/set)

~~6~~	~~2~~	~~3~~	7	4

curr.parent = 5 (Dont add because already there in set)

curr = 3

~~6~~	~~2~~	~~3~~	7	4	1

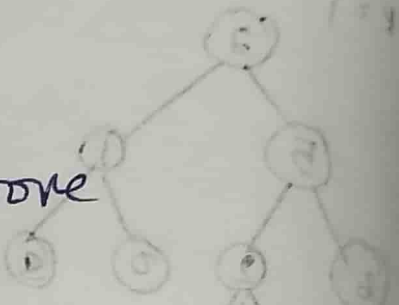
curr.left = 5 (No add, set contains already)
curr.right = 1 (Add in q/set
curr.parent = not available.

iii) k = 0
return
Because already 2 levels visited.

Now the final elements in queue are the values that are k=2th level from target (8).

Op: [7, 4, 1]

Burning Tree (GFG)



* Same as that of the above question.

* Where in both questions we travel in upward direction also apart from left and right.

* The only difference is there we travel the tree upto k-levels.

* In this sum we

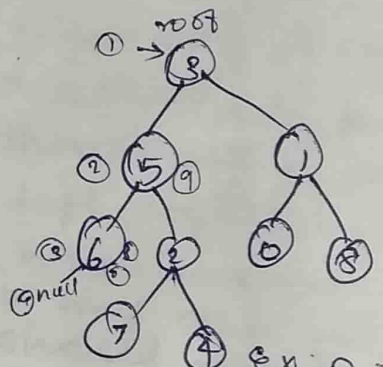
Burning Tree using DFS.

Lowest common Ancestor (LCA)

Leetcode: 236

First find p & q's traversal from the root node. Use backtracking to trace the elements

- Use Array list to add elements starting from root node till you reach the end null. If you find in either left or subtree return true to previous call else return false and also remove the last element added in the list



ex: p = 4

① [3]

② [3, 5]

③ [3, 5, 6]

④ null so return false

⑤ [3, 5, 6]

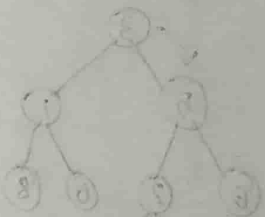
⑥ Check right of 6

⑦ null so return false

⑧ remove [3, 5]

⑨ Move to right of 5
[3, 5, 2]

⑩ Move to left of 2
[3, 5, 2, 7]
Left & right of 7 is null so return false & remove 7) [3, 5, 2]



⑪ Add 4 which is right.
[3, 5, 2, 4]
we found 4 so return list.

```
public ~~List~~ boolean backTrack
        (TreeNode root, TreeNode node,
                        List<TreeNode> list)
{   if (root = null) return
                        false;

    dist.add(root);
    if (root == node) return true;
    if (findNode(root.left, node, list))
    {
        return true;
    }
    if ( // return true;   (root.right, to 4)
    arr.remove (size-1)
} ret false;
```

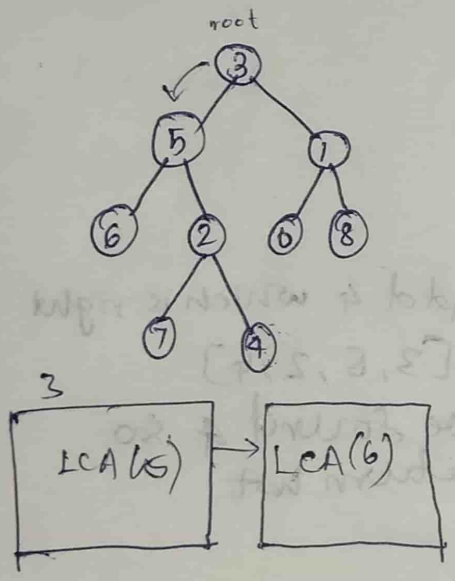
p = 5 q = 4

(5) list 1 = [3, 5]

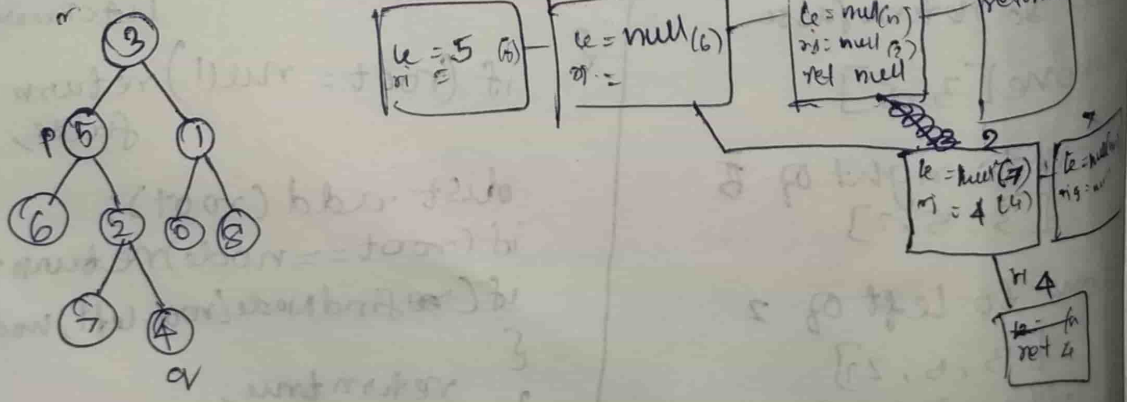
(4) list 2 = [3, 5, 2, 4]

```
for (int i = list1.size() - 1 ; i >= 0 ; i--)
{
    if (list2.contains (list1.get(i)))
    {
        return list1.get(i);
    }
}
return null // for case if no common present.
```

Using optimized Approach

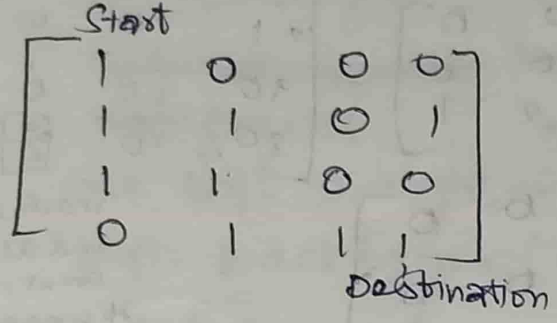


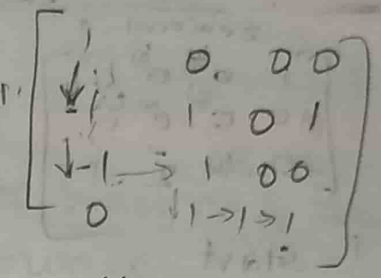
consider this example.

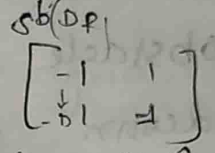


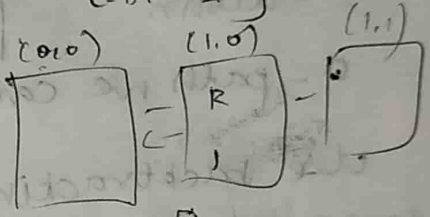
26/6

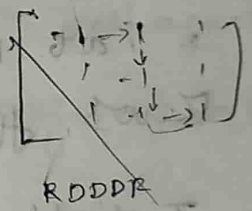
Rat in a maze:

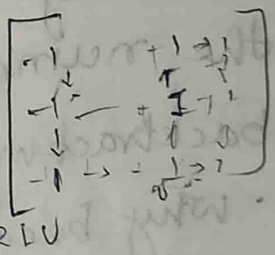












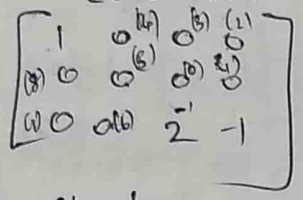
hashnode.dev/rat-in-maze-2

Start from 0,0
move to L, R, U, or D where value is 1.
When you first time visit a cell mark it as -1
then traverse through left, right, U, d because when you traverse up, left or any other direction, it should not visit already visited cell in the current path

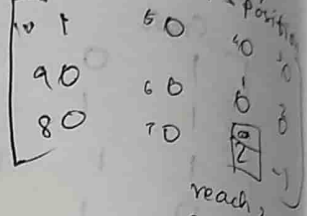
Once completing all the traversal direction make it back to 1 itself because again the present cell can be visited by its previous cell in different direction combination.

Unique Paths III LC - 980

zeroCount = 9



Actually start reducing count from 1's position



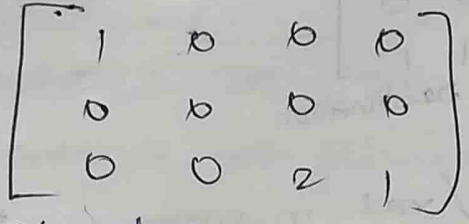
reach 2 and also count == 0 it means visited all zeros

1 - start

-1 - obstacle

2 - destin

0 - path we can travel



Use backtracking to start from 1 and reach 2 ; at note: you have to cover all the zeros. then you can consider the path valid.

Initially in the main function itself I count the number of zeros and pass to the backtracking function with zeroCount + 1 as param - why because we start backtrack function with (1's position) we will reduce zeroCount-- from that itself so param has zeroCount + 1.

```
public int backtrack(int i, int j, int count,
                     int[][] grid,
                     int m, int n)
{
    if (i<0 || j<0 || i>=m || j>=n ||
                grid[i][j]==-1)
    { return 0;
    }
    if (grid[i][j]==2 && count==-0)
        return 1;
```

int temp = grid[i][j];

~~int~~

grid[i][j] = -1;

~~int totalpaths~~ = 0

~~int~~ count--; ⟶ // to track no of zero visited

int left=0, right=0, up=0, down=0;

left = backtrack(i, j-1, count, grid, m, n)

right = "

up = "

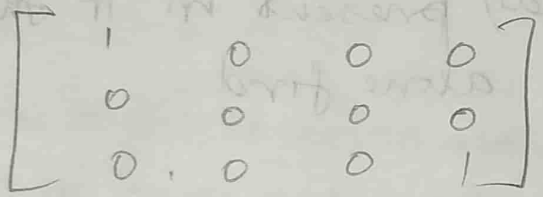
down = "

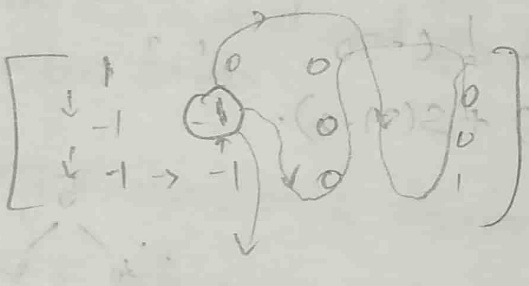
// After visiting all directions of cell again making it ~~zero~~ temp (ie its original value)

~~return~~

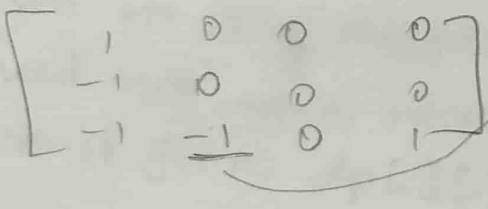
grid[i][j] = temp;

return left + right + up + down;





consider we are here We move up, right, down, left, ~~while~~ while each of the traversal you must not visit already visited so -1. After checking all back to orig

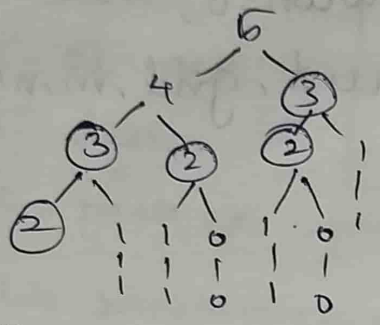


why because from here it can move R U L and also U so

Dynamic Programming

Fibonacci Number

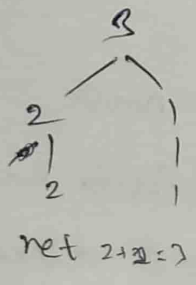
If n = 5

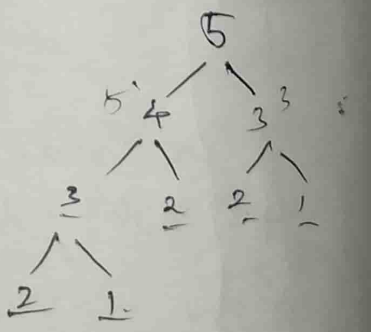


Here 3, 2 all are repeating but when using recursion we have to find each thing again, due to this extra time consump.
To resolve we use dyn prog, where we initialize dp array and store the value in dp[] if already present in it then return or else alone find.

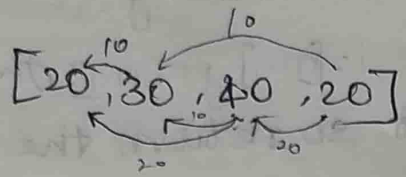
Climbing stairs LC-70

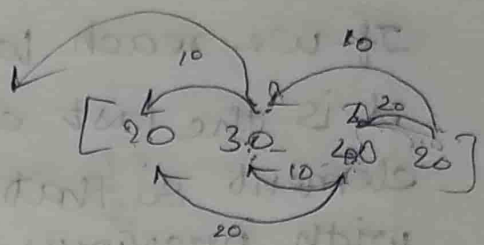
$$S(n) = S(n-1) + S(n-2).$$

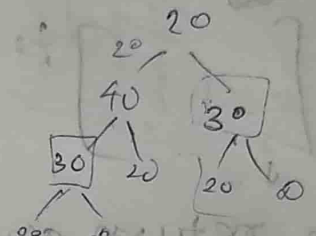




Frog Jump [GFG]







27/6

Minimum Path Sum [LC-64]

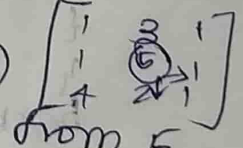
start ↓

1	3	1
1	5	1
4	2	1 ← end

Move right or down alone at a time. Within start and end find the path with minimum sum.

Using Recursion & Dynamic Progr.

Because consider you are at 5 (1→3→5)



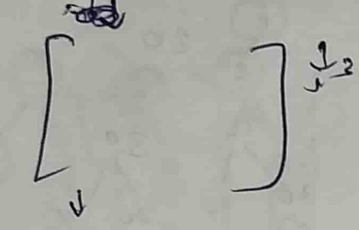
You will find the minimum path from 5 either in left path or right path. So you store dp[i][i] = val.

Again in any different path for ex: 1 → 1 → 5 you need not again recurse for 5, instead return it from dp[][] array

First,

Fix the base conditions correctly.

If we reach last cell i=[m-1] j=[n-1] then it is the last element so return the element so that it would be summed with previous one.

If  i, j goes out of index

then return any max value so that when comparing with min value this path would be rejected.

```
dp[i][j] = grid[i][j] + Math.min(
                    Solve(i+1, j, grid, dp, m, n),
                    Solve(i, j+1, grid, dp, m, n))
```

Another approach using void recursion approach.

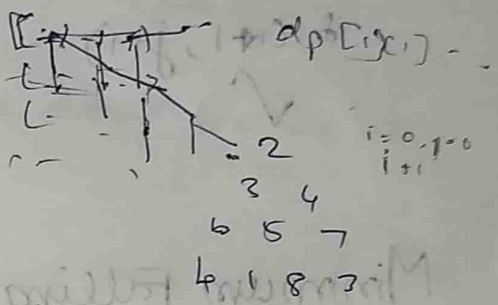
```
int minSum = Int.max
void backtrack(int i, int j, grid, dp, m, n,
                                        currSum)
{
    if (i >= m || j >= m)
        return;
    if (i == m-1 || j == n-1)
    {
        currSum = currSum + grid[i][j]
        minSum = Math.min(minSum, currSum)
        return;
    }
    backTrack(i+1, j, ....)
    backTrack(i, j+1, ....)
}
```

Triangle - Min path sum is the output

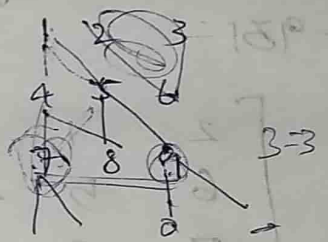
Leetcode: 120

[[2], [3,4], [6,5,7], [4,1,8,3]]

2
3 4
6 5 7
4 1 8 3

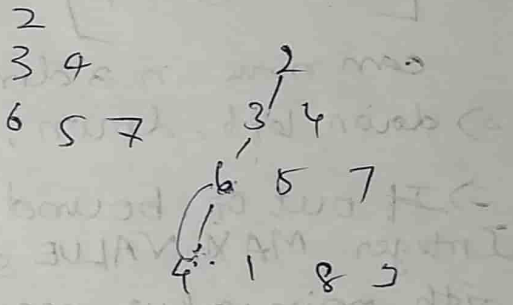


This is same as the previous sum, same approach.



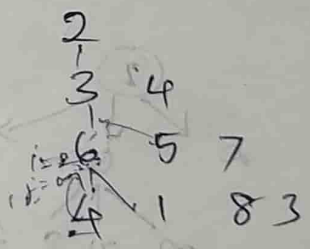
One main difference is there we recurse like

```
sum = grid[i][j] +
Math.min(rec(i+1, j, ....),
         rec(i, j+1, ....))
```



In this sum we have to use like

```
sum = list.get(i).get(j) +
  Math.min(rec(i+1, j, ....),
           rec(i+1, j+1, ...));
```

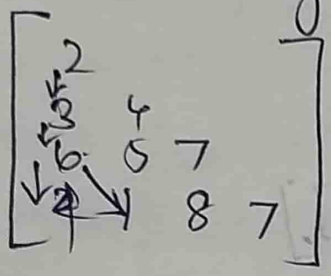


↓

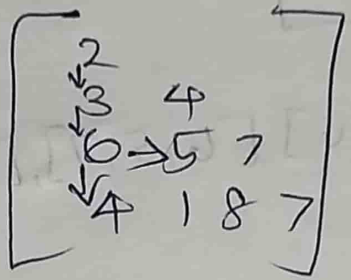
Because in prev sum, it is matrix [" " " / " 5 → Right / " ↓ Down "] after finding down dive we move right.

But here [2 / 3 4 / 6 5 7 / 4 1 8 7] after finding down of 6 (ie. 4) we have to move down right (ie. 1)

Instead of (i+1, j+1) if we use (i, j+1) we would again travel in the same level



if (i+1, j+1) ✓



if (i, j+1) ✗

Minimum Falling Path Sum

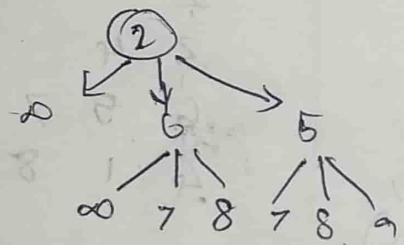
LC-931

$$\begin{bmatrix} 2 & 1 & 3 \\ 6 & 5 & 4 \\ 7 & 8 & 9 \end{bmatrix}$$

can move in 3 direction

⇒ down-left, down, down-right

⇒ If out of bound return ∞ that is Integer.MAX_VALUE so that when comparing with min value, maxval would be ignored

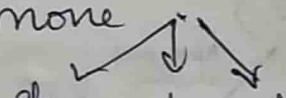


This is same as above approach but one change we made here is filled dp[][] array with

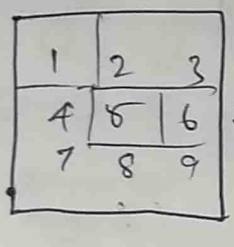
Minimum Falling Path Sum II

LeetCode: 1289

In MFPS I we move



In this sum, we move down & where below index != curr index



If we are 1 we move to 5 & 6 not straight down to 1.

So ~~we~~ in previous code in the recursive function we statically call 3 times down left, down, down right.

* But here we dynamically move down using a for loop starting from 0 to n-1 ignoring just the index below which is equal to current index.

Subset Sum Problem (GFG)

① arr[] = [3, 34, 4, 12, 5, 2]
(0, 1, 2, 3, 4, 5)
sum = 9

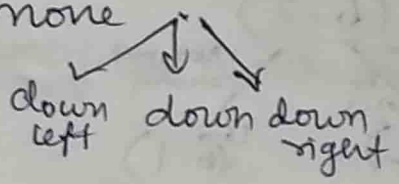
9

```
public static boolean helper (boolean[][] dp, int[] arr,
                              int sum, int idx)
    if(sum==0)
      return true;
    if(idx<0 || sum<0)
      return false
    if(dp[idx][sum]!=null)
      return dp[idx][sum];
    boolean exclude = helper(dp, arr, sum, idx-1)
    if(arr[idx]<=sum) ~~bool include = helper(dp, sum, arr~~
```

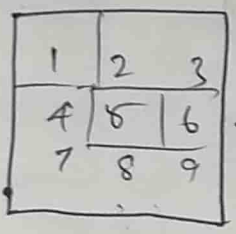
Minimum Falling Path Sum II

LeetCode : 1289

In MFPS I we move



In this sum, we move down & where below index != curr index



If we are 1 we move to 5 & 6 not straight down to 1.

So ~~we~~ in previous code in the recursive function we statically call 3 times down left, down, down right.

* But here we dynamically move down using a for loop starting from 0 to n-1 ignoring just the index below which is equal to current index.

Subset Sum Problem (GFG)

① arr[] = [3, 34, 4, 12, 5, 2]
sum = 9.
0 1 2 3 4 5

9

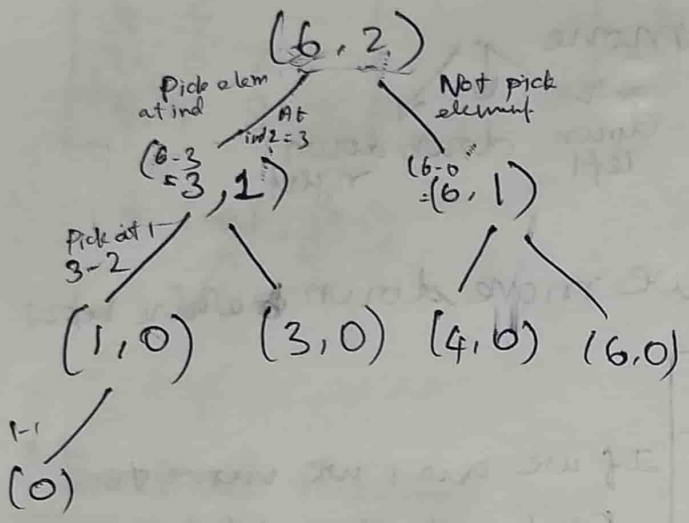
```
public static boolean helper (boolean[][] dp, int[] arr,
                              int sum, int idx)
    if(sum==0)
      return true;
    if(idx<0 || sum<0)
      return false
    if(dp[idx][sum]!=null)
      return dp[idx][sum];
```

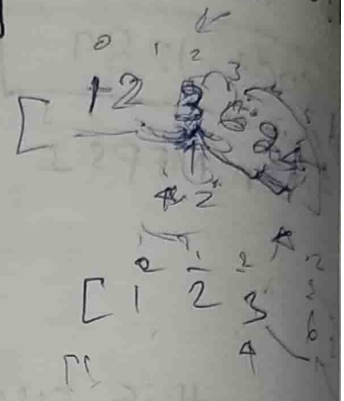
{ include = helper (dp, arr, sum - arr[idx], idx-1);
return dp[idx][sum] = include || exclude; }

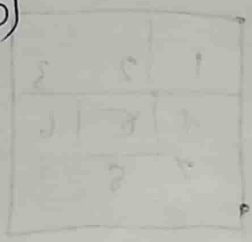
[1, 2, 3]
0 1 2

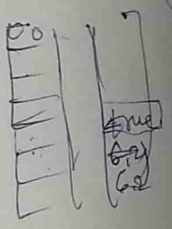
[target rem, index]

target = 6.





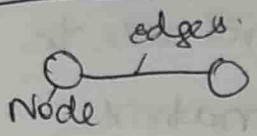




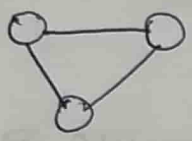
Knapsack Problem

(GTFG)

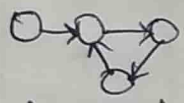
Graph

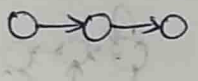


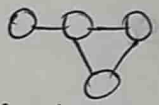
In Graph we can have cycles unlike trees.

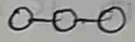


If cycle then cyclic graph.


Directed Cyclic


Directed Acyclic


Undirected cyclic


Undirected Acyclic

"All trees are ~~not~~ graph, but all graphs are not trees"

Adjacency Matrix for graph

The elements are 0, 1, 2, 3

So creating adjacency matrix.

	0	1	2	3
0	00	01	02	03
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

Time complexity for creating

adj mat : $O(n \times n)$

space complexity : $O(n \times n)$

Finding an element : $O(1)$ → can easily find with index.

Deleting an element : $O(1)$

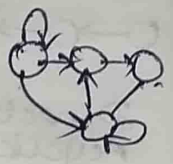
dfs : $O(n^2)$

Sparse graph:

In the above graph we are not able to utilize the complete space in the matrix. So it is a sparse graph

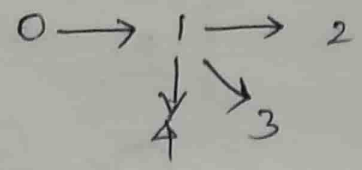
Dense graph.

If all the space is used because of all node connected to each other and it is then it is dense



```
public class Main {
    psvm
    {
        int v=5   // 1 value plus than node
        int[][] edges = {{0,1},{0,2},{2,3}};
        int[][] matrix = new int[v][v];
        for(int[] edge: edges)
        {
            int u = edge[0];
            int v = edge[1];
            matrix[u][v] = 1;
            matrix[v][u] = 1; // if undirected graph
        }
    }
}
```

	0	1	2	3	4
0		1			
1	1				
2				1	1
3			1		
4			1		



Directed Acyclic graph.

Number of node → 5

Adjacency list method

0	{1}
1	{2, 3, 4}
2	{ }
3	{ }
4	{ }

Time complexity:

O(E+V)

E - number of total edges in graph

V - Number of vertices in graph.

O(5+4)

Space complexity:

O(E+V)

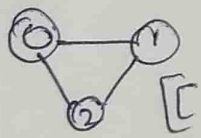
Finding an edge: O(outdegree)

[we would be given u then we have to find edge v if it exists, so it is O(outdegree(u))

Deleting an edge: O(outdegree)

Find if Path Exists in Graph

Leetcode: 1971



[[0,1],[1,2],[2,0]]

First fill the adjacency list for the graph

List<List<Integer>> list=new ArrayList<>~~Integer~~();

0-1
1-2
2-0

0 - [1]
1 - [0,2]
2 - [0,1]

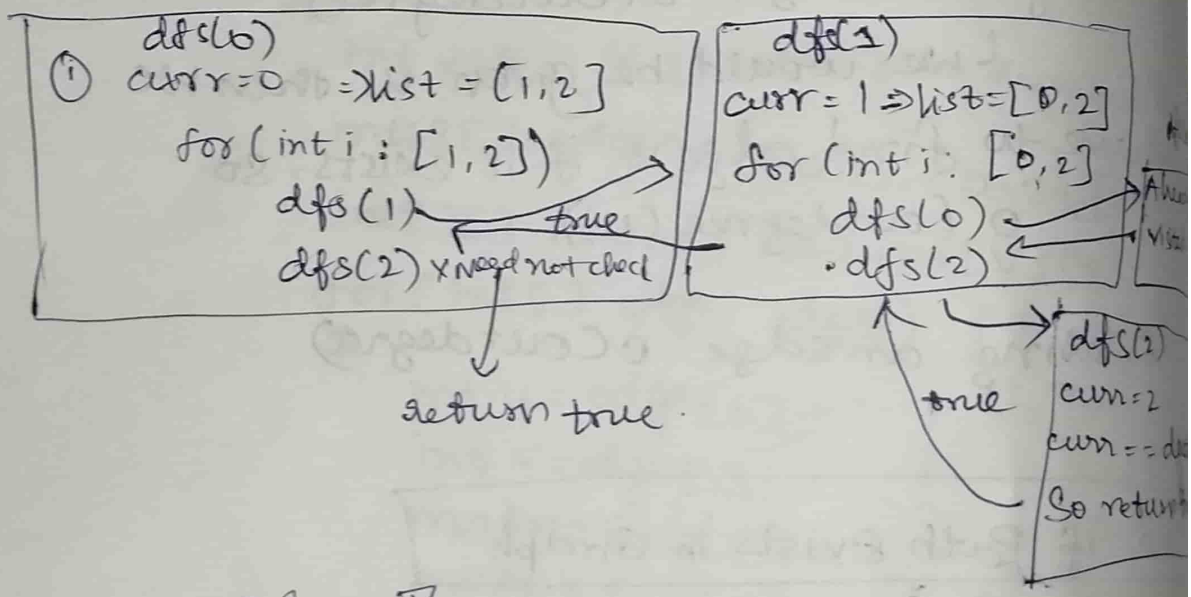
Start dfs on from source.

0
1
2

i) First select the current element.

ii) Then store the current element's neighbors in a list and iterate through them to perform dfs.

0 - 1, 2
1 - 0, 2
2 - 0, 1



① 0 - [1], 2
1 - 0, 2
2 - 0, 1

visited=[0]

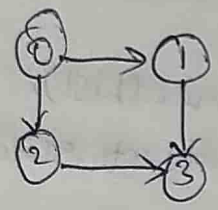
② 0 - 1, 2
1 - [0], 2
2
visited=[0,1]

③ 0 already visited

④ 0 - 1, 2
1 - 0, [2] → reached destination.
2 - 0, 1

All Paths from Source to Target

Leetcode: 797



In last sum, they gave [0,1], [0,2]. It means we can go from 0 to 1, so we built adj list.

But in this sum they gave.

[[1,2], [3], [3], []].
0 1 2 3

The each array represents the vertices traveled from the index. So directly we can perform traversal on them

Diff:

In prev

{1,0}, {1,2}

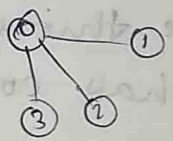
means

So we have to create list to map the vertices connected to each node.

Here in this sum

From 0 From 1 From 2 From 3

[1,2,3], [2,3], [3,], []



Here in this sum it is already in the format of list with the help of index

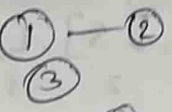
0th array means vertex linked with 0 node.

```
void dfs(int[][] graph, int current, List<List<Integer>> res,
                                        List<Integer> currList
{
    if (current == graph.length-1)
        res.add(new ArrayList<>(currentList));
```

```
else
{
    for(int i : graph[current])
    {
        currentList.add(i);
        dfs(graph, i, result, currentList);
        currentList.remove(currentList.size()-1);
    }
}
```

Number of Provinces

Leetcode: 547



isConnected = [[1,1,0],[1,1,0],[0,0,1]]

First take index 0 that is ①

① -> [1, 1, 0],

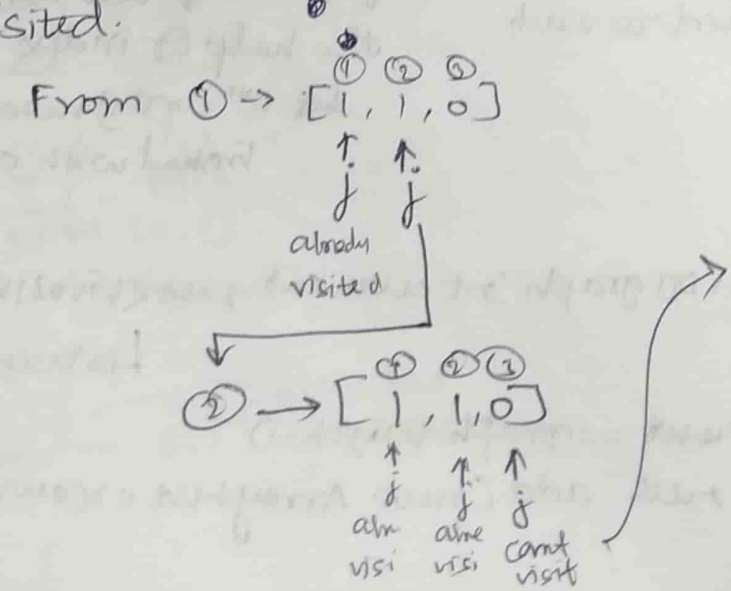
it means it is with 1, 2 but not 3.

~~So if~~

So iterate through vertices of node where it has connection [i][j] -> 1 means connected

perform dfs until you visit all nodes.

Each time visit and mark the node as visited.



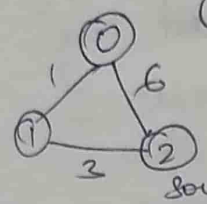
Now again move to next iteration in main function for loop i=1. now => which denotes node ② => already visited.

So move to ③ perform dfs increement count of province.

So total output: 2.

Dijkstra

(GFG) (yasamitsingh.hashnode.dev/dijkstra)



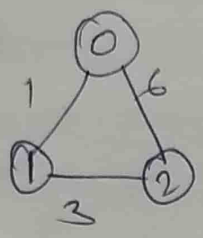
int[][] edges = [[0,1,1], [1,2,3], [0,2,6]]

Src = 2.

OP: [4,3,1]

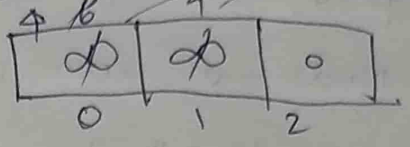
- First form the adjacency list for the given question.
- We need 3 datastructures to use here.
  - List<List<Int>> → To create adjList
  - PriorityQueue<Pair>
    - ↳ It is used to add the next nearest distance which stores value in form of Pair class.
    - ↳ We create PriorityQueue<> to return the lowest distance element in front of queue. So that we can poll that and travel from that node
  - Dist arr[], which stores the least distance in each ~~array~~ index from the source,

Only if we discover a node with less distance from source then we change the dist[] arr value in its index and add the (node, weight) value in Priority queue.

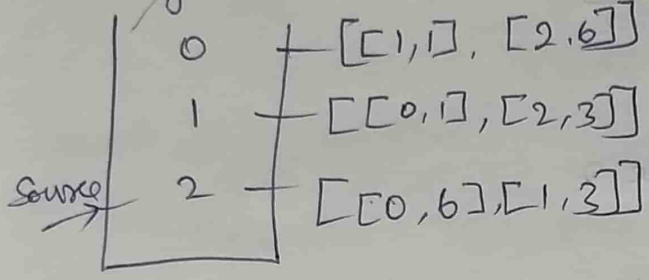


Dist[] arr

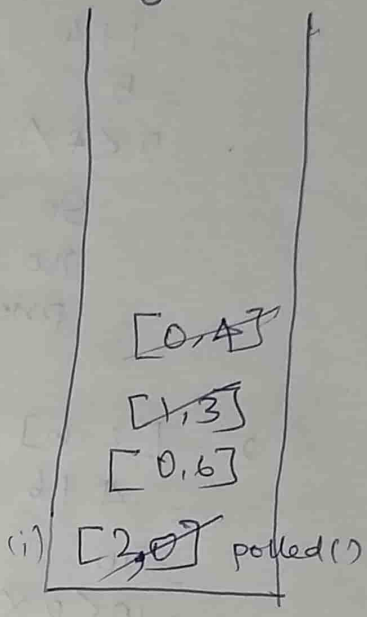
Initially infinity so we store min dist while tra



Adjlist



Priority Queue



(i) queue.poll [2,0]

- Node-2 [curr weight-0] → It is the total weight travelled by.

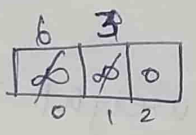
From 2 traverse to

0 & 1

0 - current weight + Node(0) weight

- 0+6 = 6

6<∞ ✓



1 - cw + w

- 0+3 = 3

3 < ∞ ✓

(ii) Now poll the value with smallest distance.

queue.poll [1,3]

Node-1 [current weight-3]

From 1 traverse to 0 & 2

0 - (0,1)

= 3+1 = 4

4<6 ✓ [6 | 4 | 0]

2 - (2,3)

= 3+3 = 6

= 6<4 ✗ [4 | 3 | 0]

Push the new smallest distance along with the node.

Here new dis - 4

node - 0

(iii) [0, 4] queue.poll()

Node - 0, curr weight - 4.

Move to 1 & 2

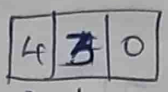
1 - [1, 1]

= 1 + 4

= 5

5 < 4 X

So no change & no push in priority queue.



2 - [2, 6]

= 4 + 6

= 10

= 10 < 0 X So no change & no push

(iv) priority queue.poll (0, 6)

Here also curr weight is high so, no changes and no new priority queue value.

Now queue is empty and the distance array is filled with nearest value.

Leetcode 743:

Network Delay Time

Same as that of the previous problem.

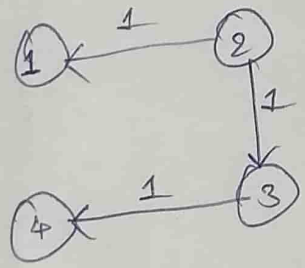
- All same steps
- we have to find minimum time taken to reach all nodes.

Consider final dist[] array

[4, 3, 1]

Here 4 is max it means the longest node is 4 seconds. So all nodes must have been travelled within that time.

- One small change is here graph



Here instead of starting node from 0 they started from 1 so create 5 empty ArrayList first itself, Leave 0th ArrayList unminded

same way create dist[] with index[n+1].