**AlgoExpert Documents**

**What is the Problem solving technique?**

First analyze the problem by writing down one input case. Find out couple of theoretical solutions in paper. I love it. Start with worst solution and slowly go to the best solution. Write pseudo code of them and find O costs. Choose the best solution and if everybody agreed then start writing code.

**Can you make any better?**

That means there is space to make it better. It is normally storing recalculating stubs in memory and reuse from there so that we don’t need to calculate again and again same things. Another technique is to store visited nodes using Boolean array or hash map would make stub lot easier.

**Problem Solving Techniques**:

1. First example problem, describe step by step very details, write theorem if you found any, write Pseudo code, then start coding, then it will be easy to implement.
2. Multi pointer approach (left + right pointer for array using **while** loop)
3. Check if **sorting** gives you any benefit, something after sorting manipulate by **while** loop and **two pointer** helps
4. Efficient use of **HashMap** with Boolean, Boolean/ Boolean, Integer/ Integer, Class Type to make it faster
5. Using **Hashmap** **duplicate** elimination, you have to put element as key and index as value
6. **Memorize** operation by two dimensional arrays: Boolean/ Integer / String / Or any Class type to **duplicate** elimination.
7. Using DFS (**Stack**), BFS (**Queue**) to manage connections or adjacent nodes
8. Using **iterative** way (while or for loop) or **recursion** (call stack: hard to debug, not memory efficient) to do similar step again and again.
9. Efficient **lambda** Stream to do many operation in Java including sorting, searching items
10. **Duplication** elimination by considering only previous elements (from current element) from array.
11. Use **Linked** list where you have to manage **4 pointers** to manage connection with previous and next node. You have to keep the chain always stable no matter what operation you do. Linked list is better when you want to insert or remove items, but for searching items this is not much faster.
12. Use **Tree** which has similar data structure like linked list (can be managed by left and right pointer). Insert and delete is easy because you can deal it from leaf node. But remove is the hardest one because you have to keep structure filled (huge rearrange) for the change, when you remove something from middle or top

**Why Tree?**

One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer: file system

1. Store hierarchical data, like folder structure, organization structure, XML/HTML data.
2. Binary Search Tree is a tree that allows fast search, insert, delete on a sorted data. It also allows finding closest item
3. Heap is a tree data structure which is implemented using arrays and used to implement priority queues.
4. B-Tree and B+ Tree: They are used to implement indexing in databases.
5. Syntax Tree: Used in Compilers.
6. K-D Tree: A space partitioning tree used to organize points in K dimensional space.
7. Trie: Used to implement dictionaries with prefix lookup.
8. Suffix Tree: For quick pattern searching in a fixed text.
9. Spanning Trees and shortest path trees are used in routers and bridges respectively in computer networks

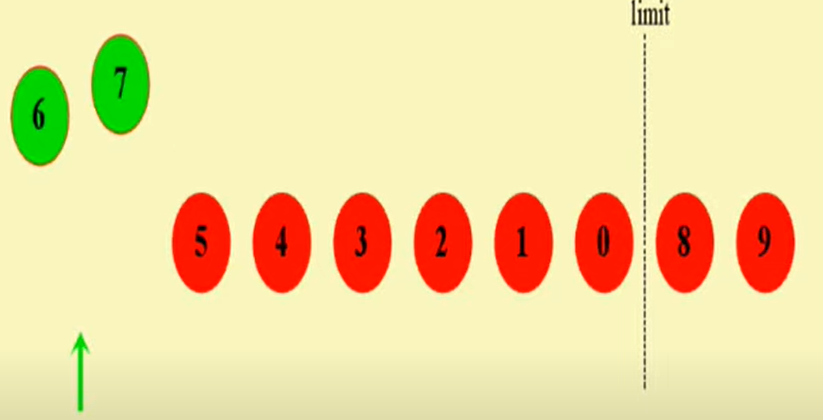
**Use case of Linked list**

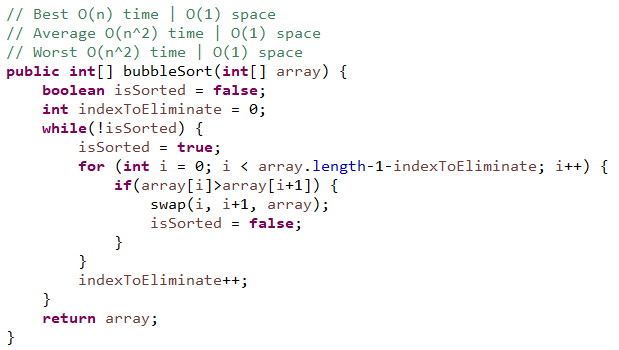
1. Implementation of stacks and queues
2. Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices is.
3. *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
4. *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
5. *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

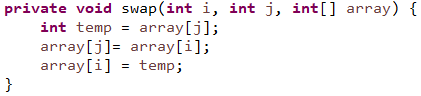
**Sorting**

**Bubble Sorting**

The **bubble sort** gets its name because elements tend to move up into the correct order like **bubbles** rising to the surface. Every time it compares with next one element, if left one is bigger, then it swaps. Every round it leaves one item from right side as a sorted part.

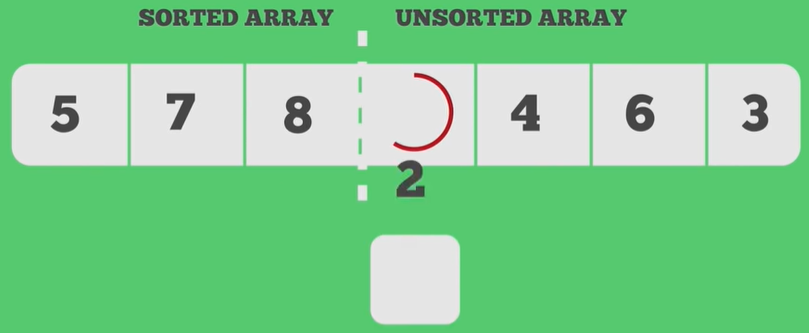
****

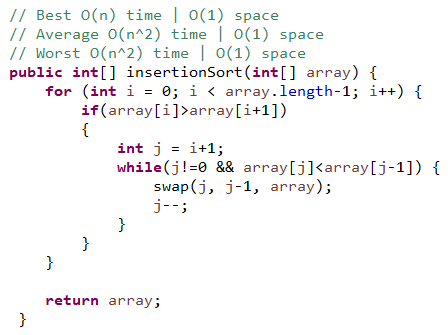
****

****

**Insertion Sorting**

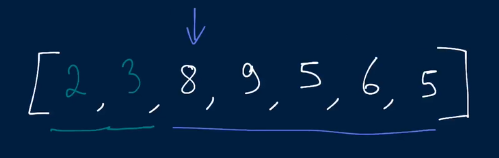
This **sort** works on the principle of inserting an element at a particular position, hence the name **Insertion Sort**. First time, it considers first element is sorted element. Then every first item from non sorted part is considered reversed order to see if it is smaller than swap it to left.

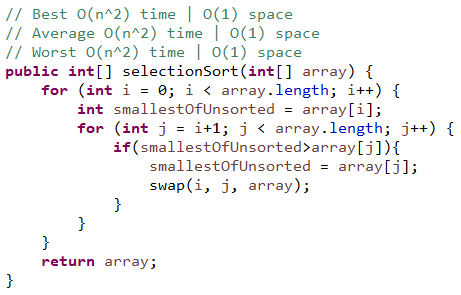
** **

****

**Selection Sort**

Selection sort is called selection sort because it always select **lowest** number and swap with first positioned number, and the select second lowest number and swap with the second positioned number, select third lowest number and swap with the third positioned number, and so on.

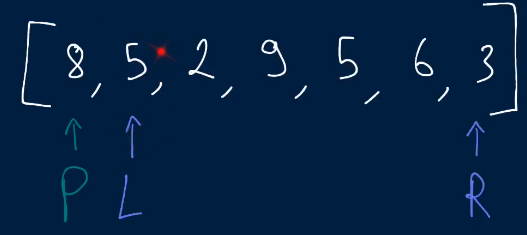
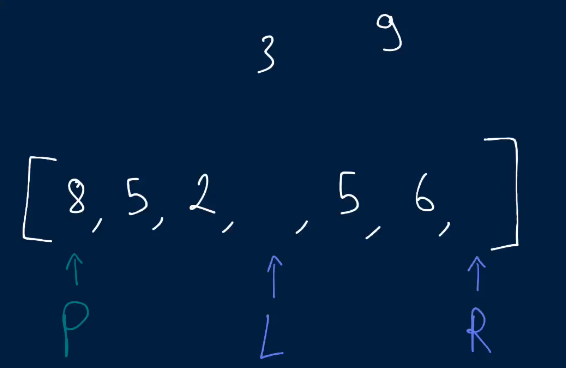


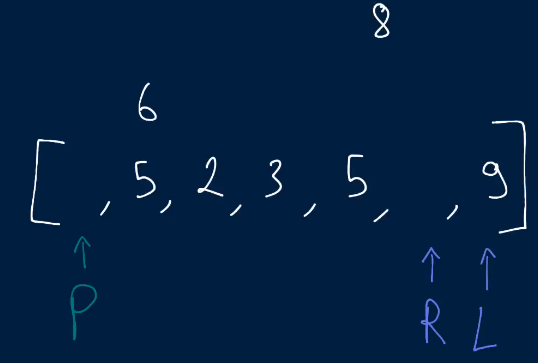
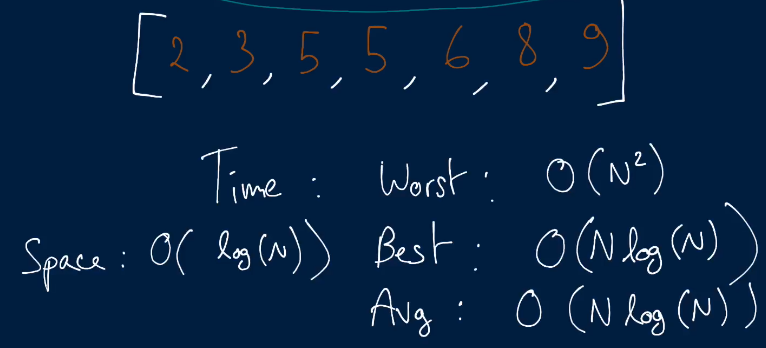


**QuickSort**

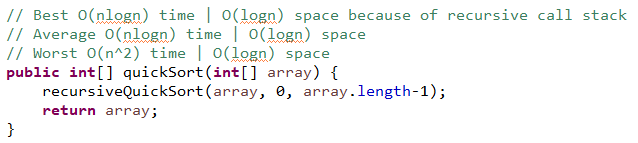
**Quick Sort** provides a fast and methodical way to sort lists of entities. It chooses an element known a **pivot** and ensures that it gets placed at its right position in the list (move smaller than pivot to left sight and bigger than pivot to right side).

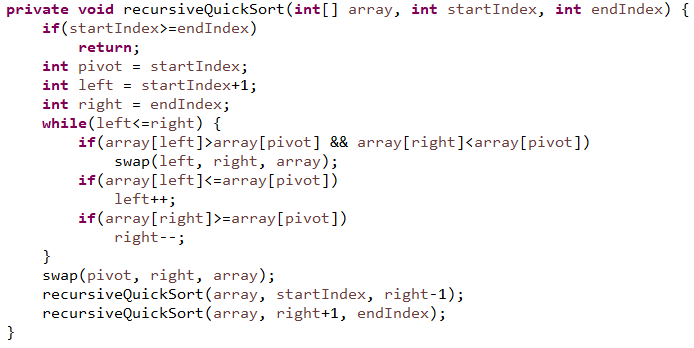
The reason is that even though it may be asymptotically slow, it works very fast in most practical cases. There is even a version which claims O (nlog(n)) average case complexity. It doesn't need much extra memory.

Our job is to select a pivot and keep numbers bigger than pivot to right side and keep number smaller than pivot to left side of the pivot. Notice the number on pivot is getting its final place at every iteration. Then provide left part of the pivot as a source of input for the recursive method and provide right part also afterwards. It does all the operation by providing the same array but by only providing pointers of the different numbers (start and end part). And it also swaps on the same array. This is quite interesting idea. The recursive method is essential here.

****

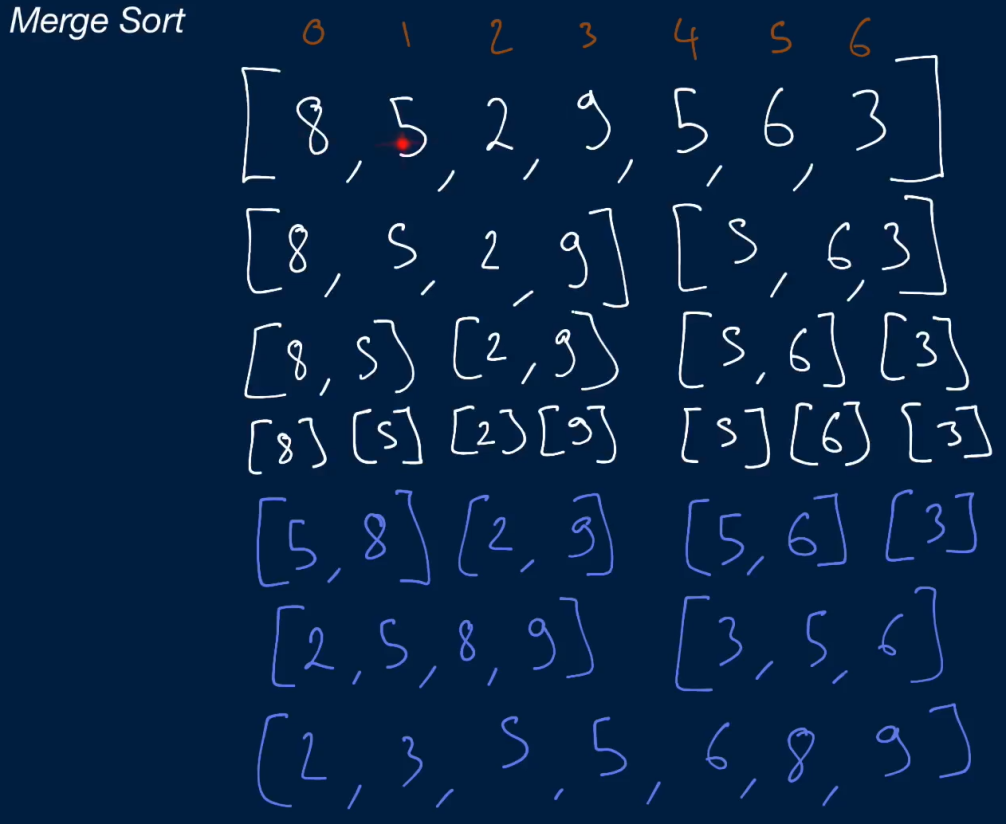
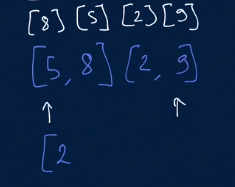
****

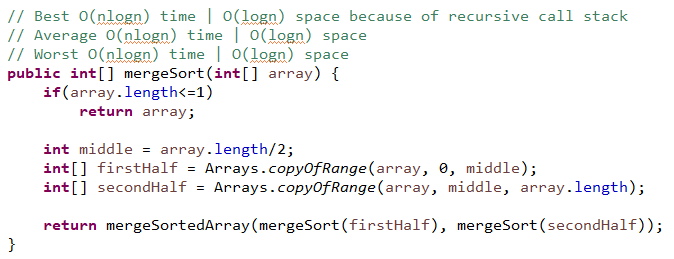
**Merge Sort**

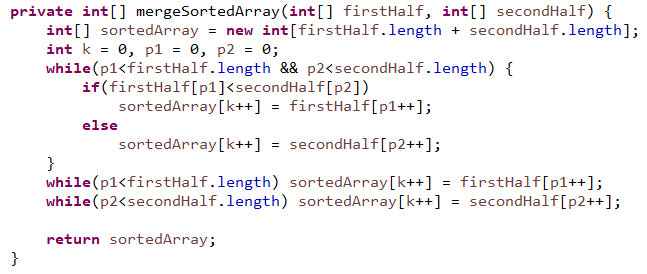
Merge sort is a divide and conquer algorithm where the idea behind is a combination of three things: dividing, sorting and merging.

**Solution 1**

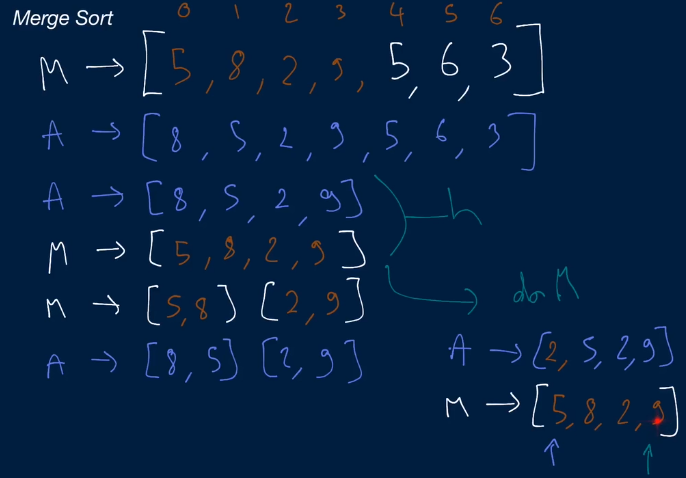
This is a beauty of recursion. For me, it was still hard to imagine using two different part recursively and calling another method at the same time by using them. Let’s think about it. For the recursive method, first of all you have to think about its exit point. It is very similar like while loop. Then you can take part of your data and with that you can do recursive call to do the same operation again. The question is what you want to do with output of that recursive call. To use the result or do further process, you can put your recursive call inside another method’s parameter. You can call that using multiple parts of these data too. To write logic on that method, you can just think about what logic you would do with first time collected data.





**Solution 2**



I have not done it yet.

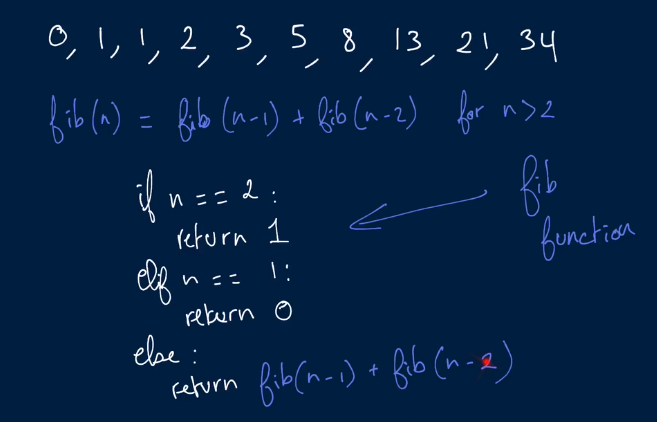
**Recursion**

**Nth Fibonacci**

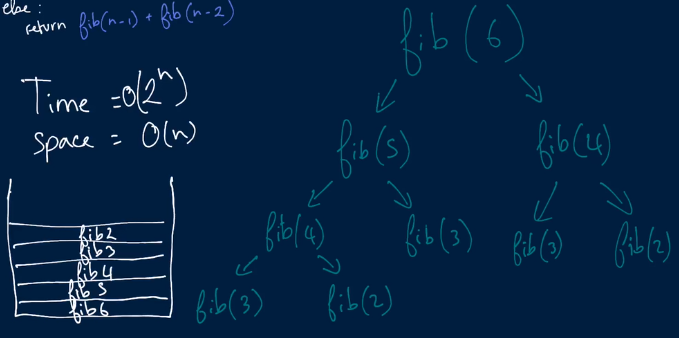


**Solution 1**

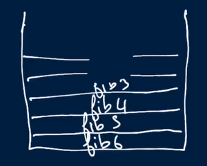
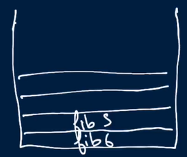
By making a recursive method call, it can do the job.

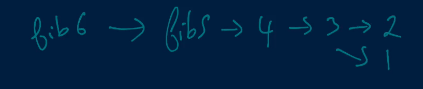


It would cost Time: O (2n) and Space: O (n). We are just making recursive call. It seems like we are not storing anything, then why it would cost space O (n)? But in recursive call, we use function call stack which uses memory. When we call Fib (6), it will generate an entry on very bottom of call stack. It realizes that it has lot of pending work. It will wait there and will generate two recursive calls Fib (5) and Fib (4) which will come on top. They are just hanging in the memory. It has to remember the state that we are on. This goes on until our base case Fib (2).



Then it will erase Fib (2) at the beginning when it is done. This is done even twice over because Fib (4), it will call first Fib (3) and then Fib (2). It will remove as LIFO order. Because we have at most n frames in that call stack, it will use O (n) space complexity.

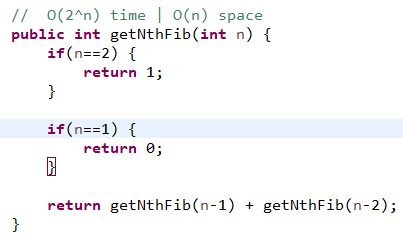
 



In the recursive method call, it generates call stack and the return is going to place from where (inside that method) it was called. It will back track the result which means one wait for another result.

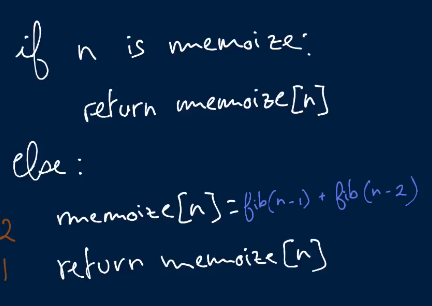
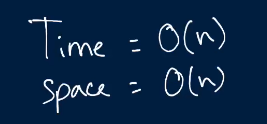
**Recursion** can be slower **than iteration** because, in addition to processing the loop content, it has to deal with the **recursive** call stack frame, which will mean more code is run, which means it will be slower and comparatively more memory use.

Solution 1 (Java Code)

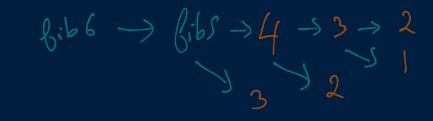
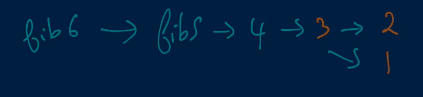


**Solution 2**

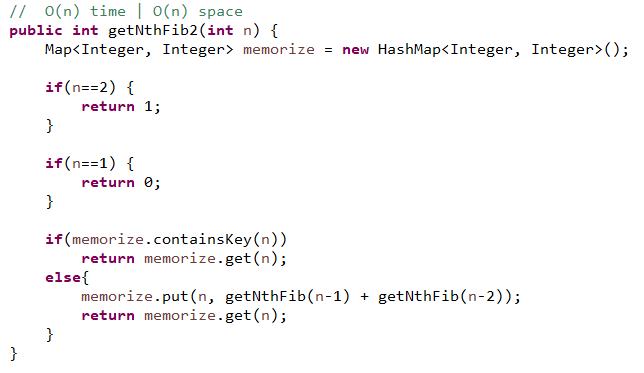
Instead of computing Fib (n), for n = 3…..n many times (same data), we will computed one in store (in Hashmap) each of the Fib (n) when a unknown Fib (n) comes. It will reduce computing complexity.

** **

Time complexity will reduce to O (n) but Space complexity will remain same as O (n) because we still use same call stack for recursive. Additionally we introduce a new Hashtable variable. But it will not increase space complexity.

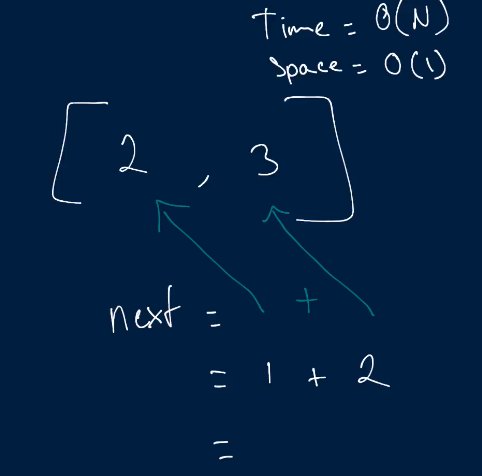
****

Solution 2 (Java Code)

****

**Solution 3**

This one is the best solution for this problem because it uses only a two dimensional array to solve the problem which is almost nothing. It is just shifting the next value from right to left and the right side array is interchanged by the result of adding current two values. This is an amazing solution which costs O (n) time and O (1) space complexity.

** **

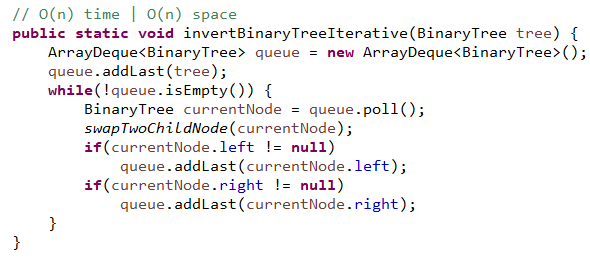
**Binary Tree**

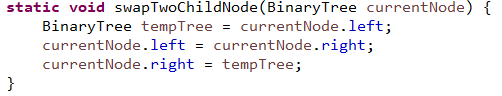
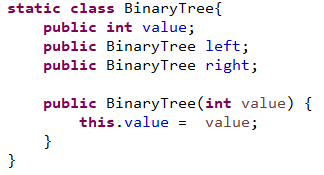
**Invert Binary Tree**

****

**Solution 1 (Iterative Solution)**

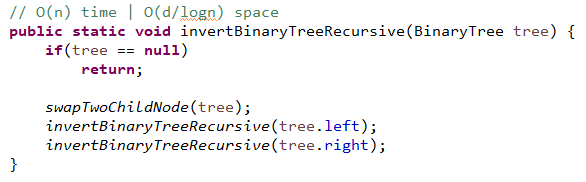
This is an elegant work with iterative solution and I like it because it works with queue and FIFO. It picks a node and handles its Childs. It is very easy to understand solution. But the recursive solution has a bit better space complexity.

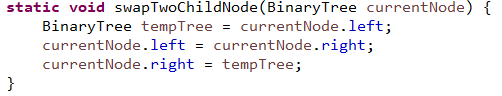
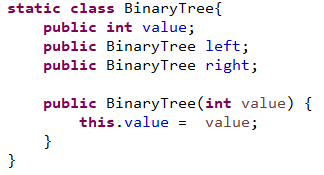
****

** **

**Solution 2 (Recursive Solution)**

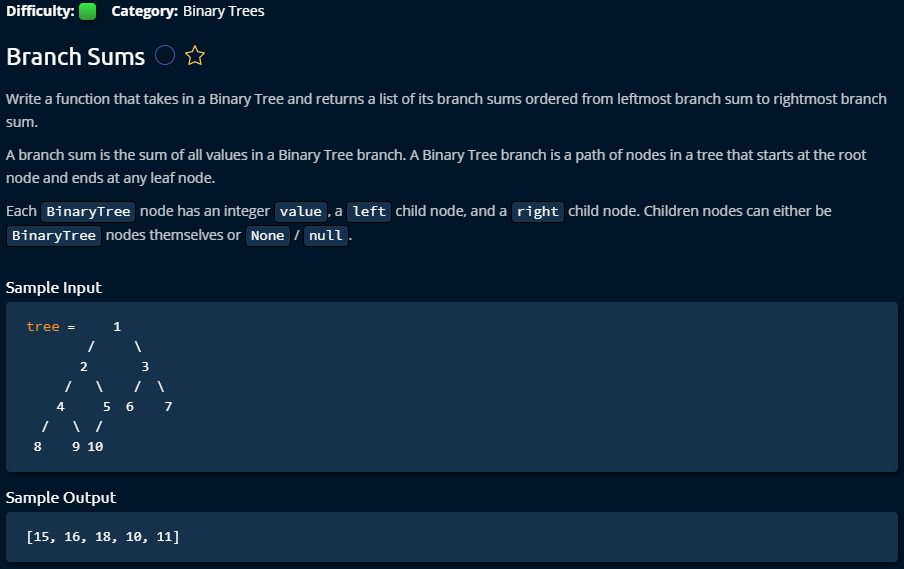
This recursive solution has a bit advantages over other iterative solution. Due to the nature (DFS like) of recursive call, it needs to store call stack but it will always go the depth of tree to the left side first and next right. So, we don’t need extra queue to keep track of data but the call stack has to be remembered. And the call stack has to remember until depth and it will go from depth to next right side, so it will make the call stack free from left side as it completing the operation from left depth. So, the space complexity we need until depth which means logn.

****

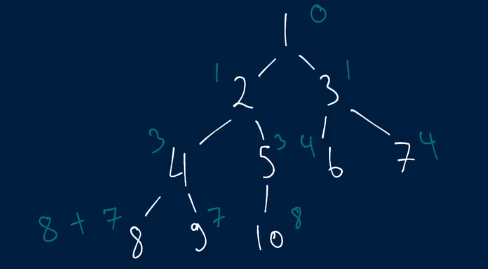
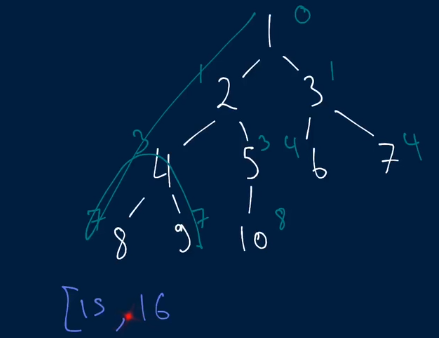
** **

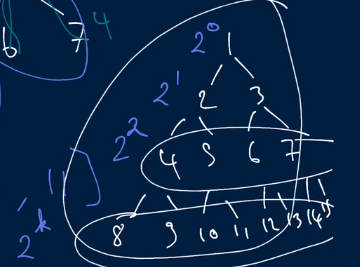
**Branch Sums**

This is a recursive solution which execution style is similar like DFS algorithm. It deals with call stack. Due to the nature (DFS like) of recursive call, it needs to store call stack but it will always go the depth of tree to the left side first and next right. So, we don’t need extra queue to keep track of data but the call stack has to be remembered. And the call stack has to remember until depth and it will go from depth to next right side, so it will make the call stack free from left side as it completing the operation from left depth. So, the space complexity we need until depth which means logn. But we need to remember the summation result of each node hierarchically which also need memory, that’s why we provide a **list by method parameter** to keep track of summation result and it will only add the result to the list once it find the leaf of a side (by checking if there is no child in either left or right).

****

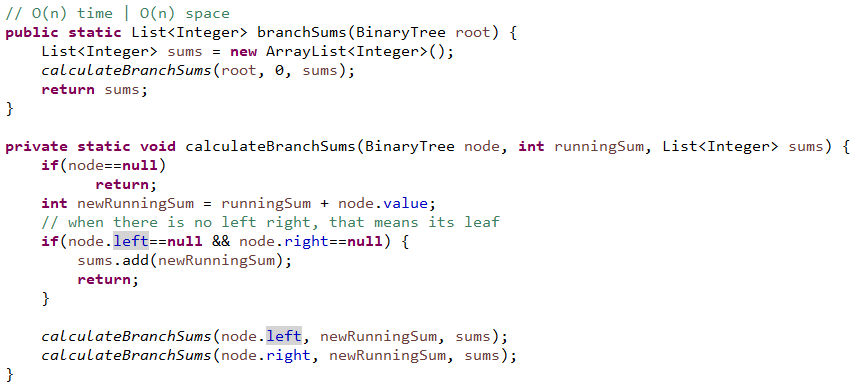
It comes to the node with remembering previous nodes upper hierarchy’s summation and sum with current nodes values.

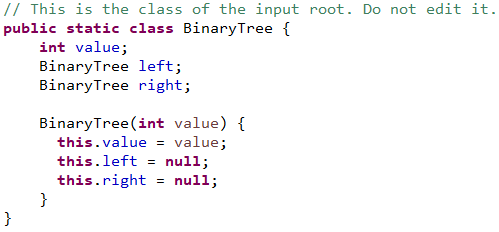
** **

**** Normally it contains 2n nodes in each row of trees.

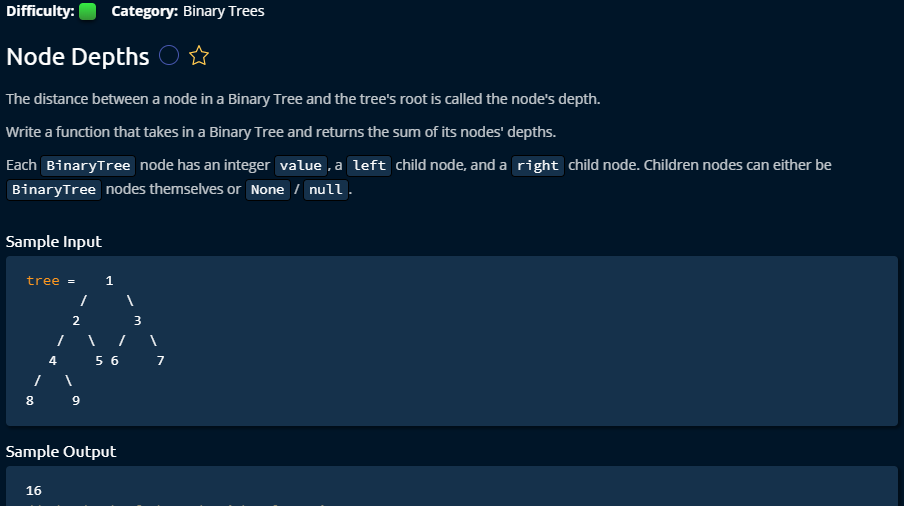
**Solution**

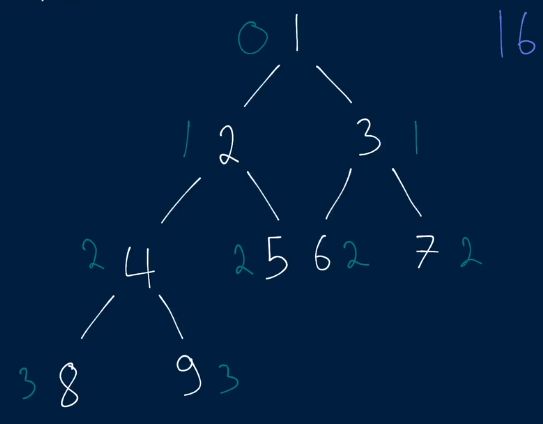
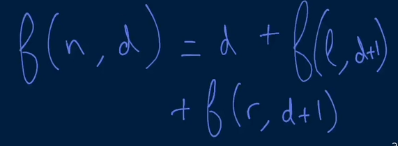
It has a recursive technique to find its each leaf. This is elegant solution.

****

****

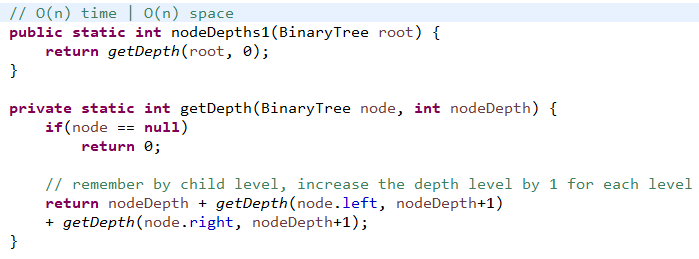
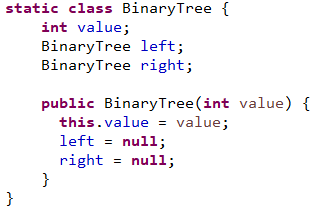
**Node Depths**

****

** **

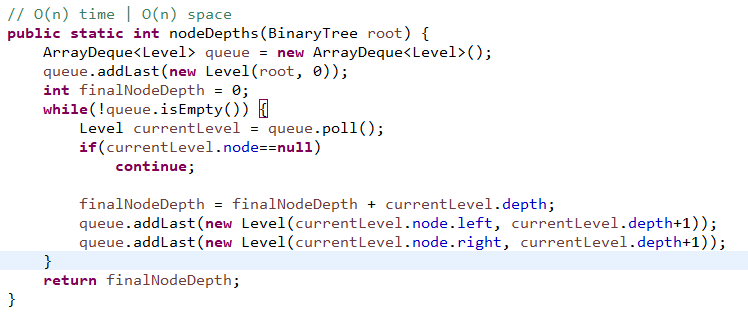
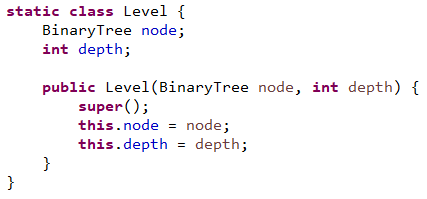
**Solution 1**

Both of these solutions have similar time and space complexity. Recursive one seems like a bit confusing to me.

** **

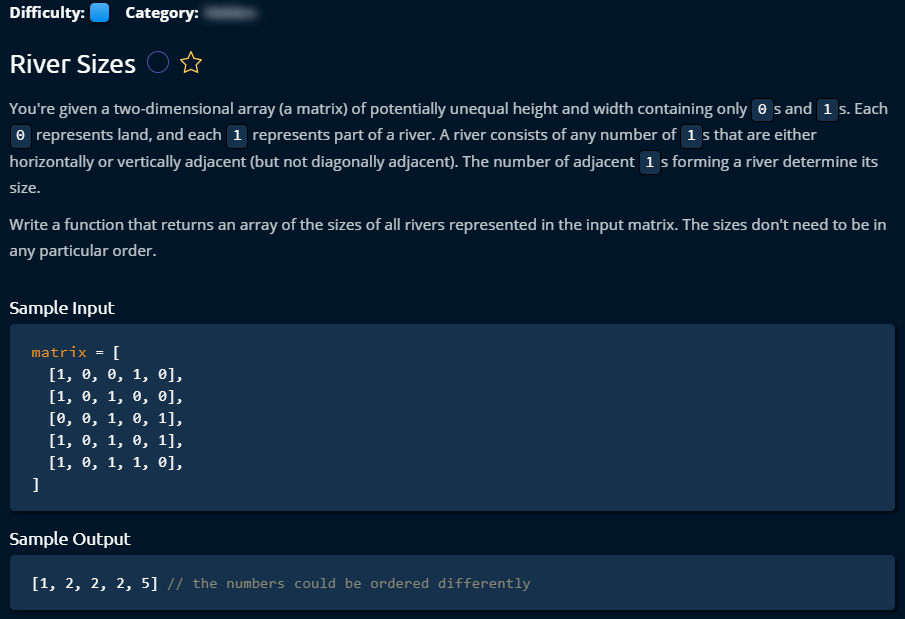
**Solution 2**

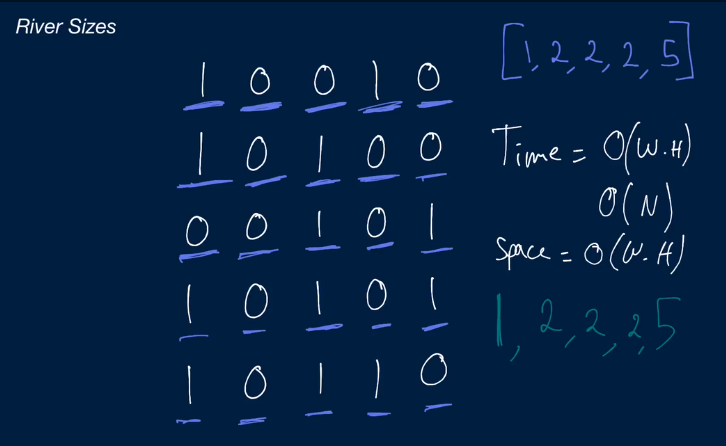
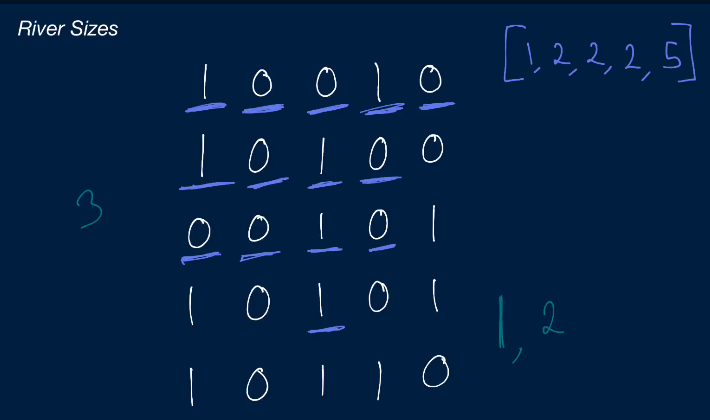
I like more this solution 2 because it works with normal queue techniques which are easy to implement and understand.

** **

**Graphs**

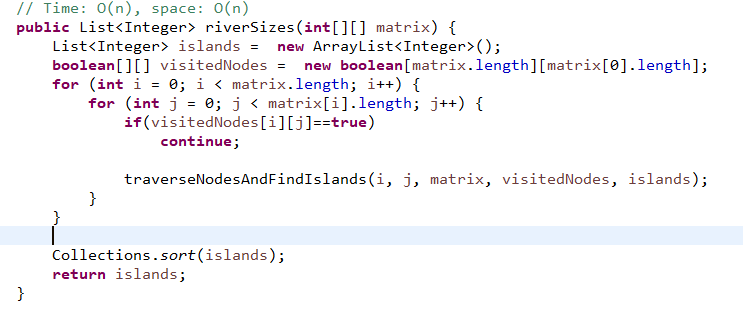
**River Sizes**

****

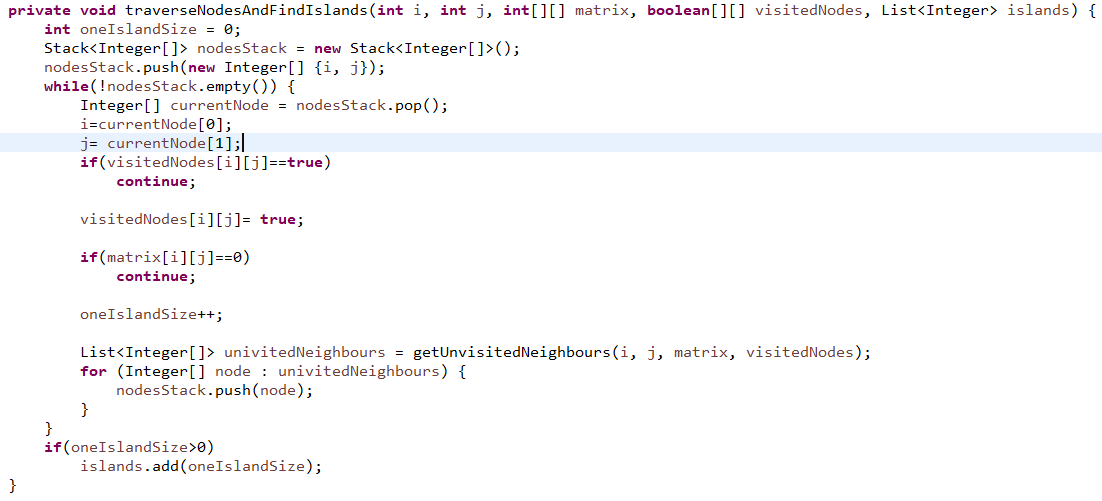
****

**Solution 1**

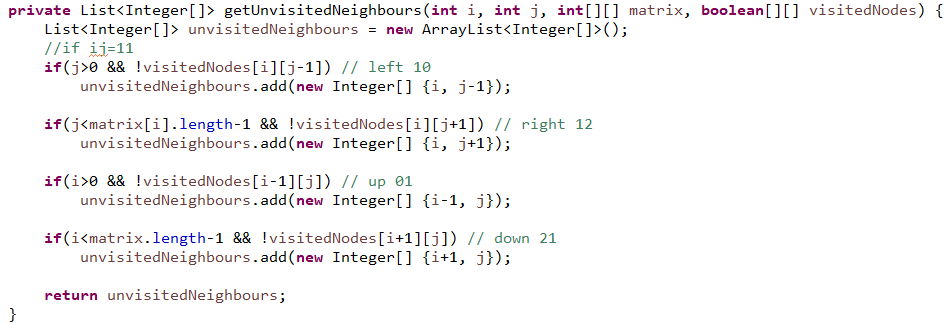
This is a very interesting problem related to graph which can be done by DFS or BFS to find all the adjacent relevant nodes. To reduce the operation times, we have to keep track of all visited nodes and make sure that we don’t visit the same node twice.

****

Starting with first node, we send it traverse all adjacent nodes using DFS, and store the first node inside the stack. We loop through all existing nodes in the Stack. When we found 0 then don’t consider and stop traversing that direction, but when we found adjacent node 1 then we increase the size of river and go further in that direction. The whole nodes are stored in Stack and consider one after another by push and pop operation.

****

All the adjacent nodes can found by following tricks, then we add them in the Stack to consider for the next round in calculation. This way we can go until end of adjacent 1.

****

**Solution 2**

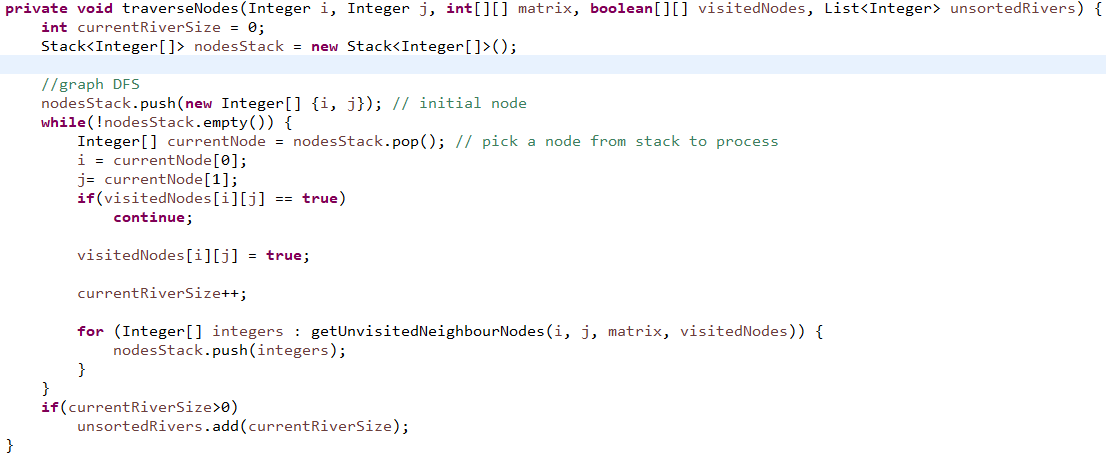
This last solution can be made a bit faster using some tricks.

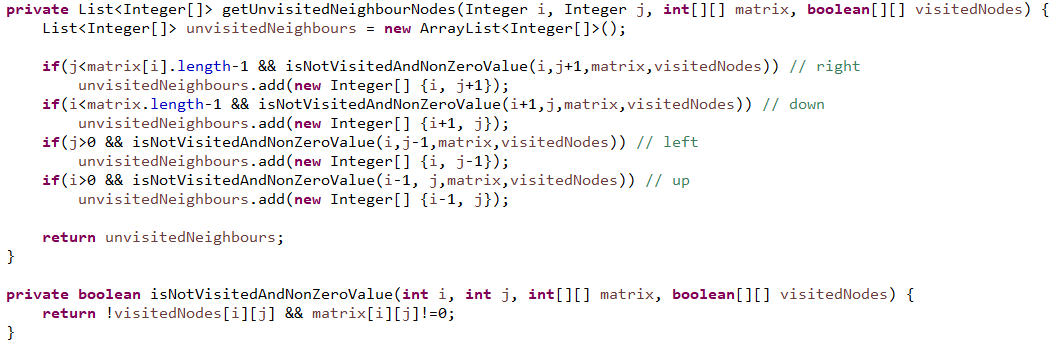
From the first part, we considered all elements 0 or 1 to find if it has adjacent 1. We only had one filter which is if the node was already visited. Now, we can also avoid 0 because 0 cannot have a river. So, we will consider only 1 nodes and if it is not yet visited. It will also reduce lot of time and space complexity.

In the last technique, in the third part, we pick all the adjacent elements of 1 (all 3 sides) even if they are 0. Now, we are going only pick adjacent nodes which are 1. It will reduce lots of time and space complexity.

We also extract a new method to make our last method little bit readable.

****

****

****

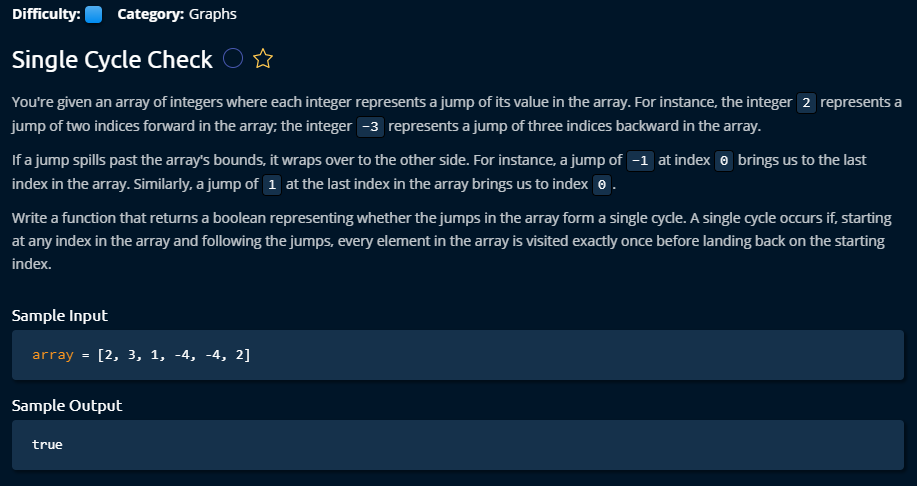
**Related problems**

463. Island Perimeter (Leetcode)

130. Surrounded Regions (Leetcode (medium))

200. Number of Islands (medium)

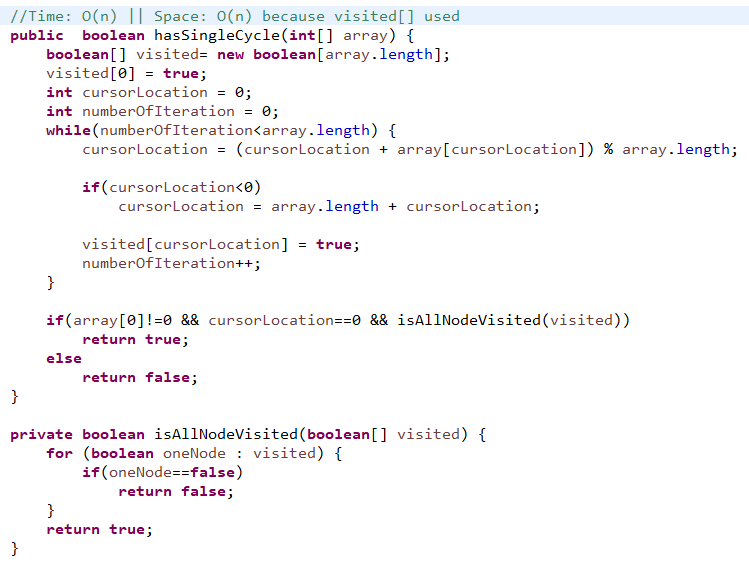
**Single Cycle Check**



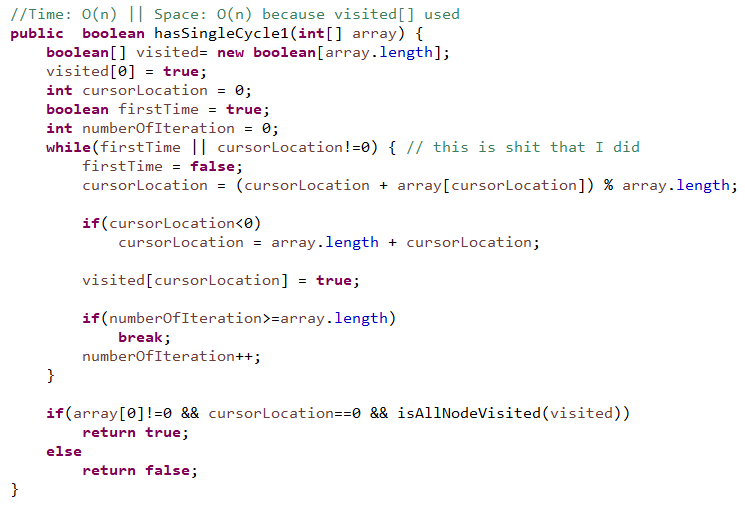
**Solution 1**

This is the solution came into my mind when I started to solve the problem. I used a Boolean array to keep track of visited nodes. As the problem was tricky and input can be wide range and the large outer bounded values had to adapt into the array, I had to do mod operation with array length. When the index was negative, then I had to make a summation with length again to keep it into the array. It costs O(n) time and O(n) space due to visited[] use. We could do the whole job even without keeping track of visited nodes which could lead to reduce the space complexity to O(1).

One thing is worth remembering that I had problem to decide the condition for while loop which should confirm that given array was considered until its length. Here is the following I also attached my first messed up copy of solution.

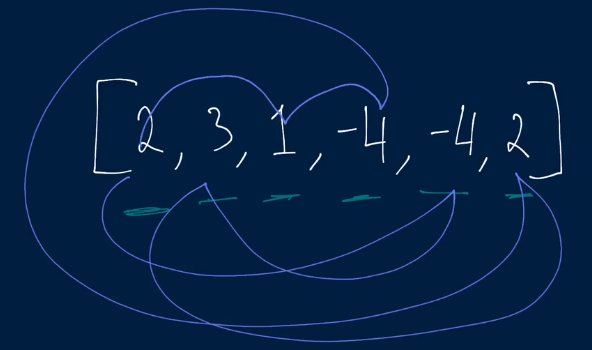
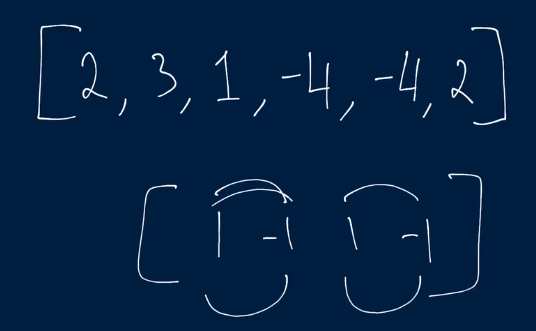


My first messed up copy of solution. I need to find a way to avoid this kind of problem.

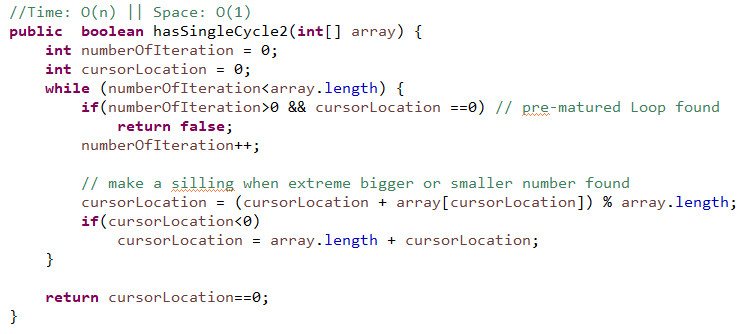


**Solution 2**

This problem could be done without remembering node visited. Only we have to care if there is any pre-matured loop, then we return false immediately. When any node is 0, that means it does not move any more. And we have to consider some corner cases.

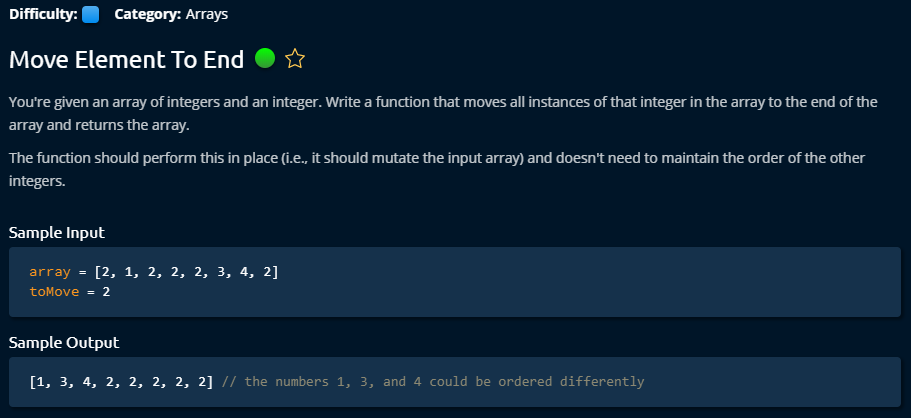
 

This is an elegant solution with very less complexity, as we don’t need any extra list.



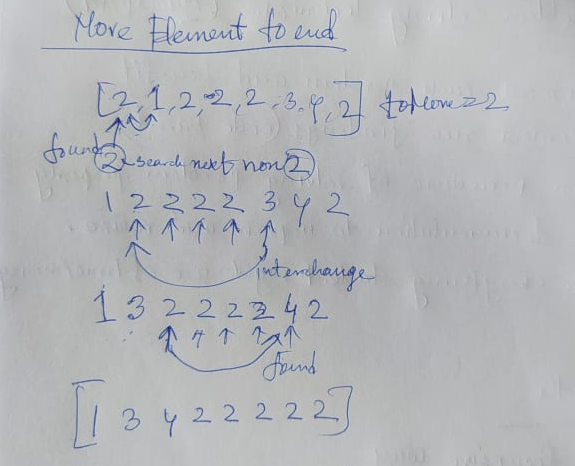
**Arrays**

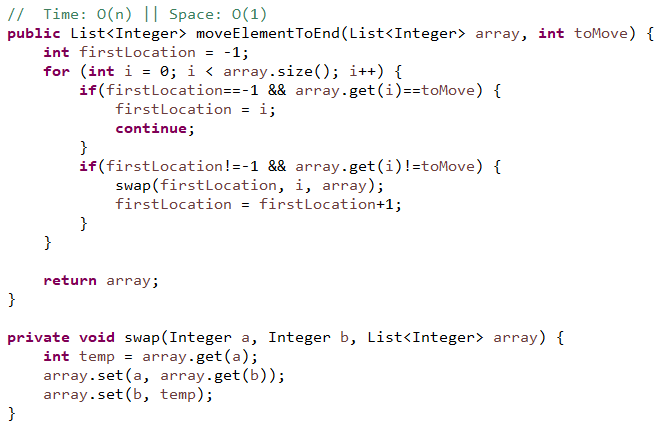
**Move Element to End**

****

**Solution 1**

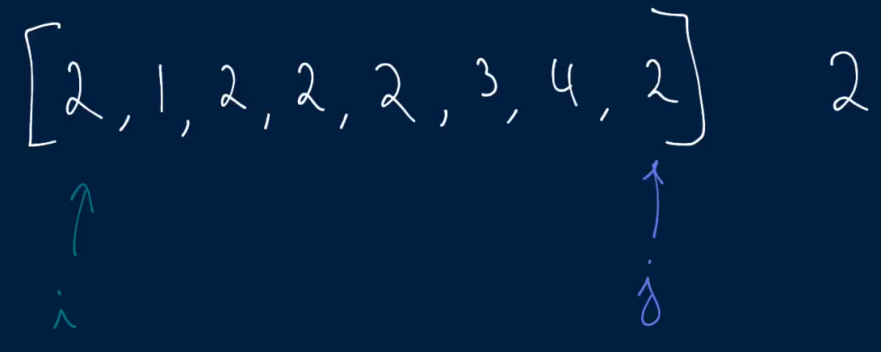
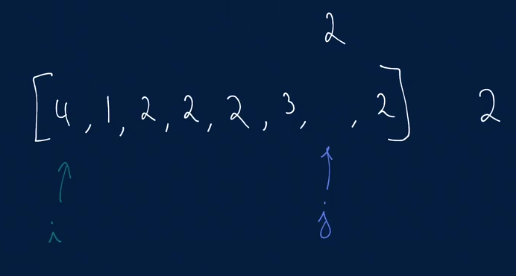
This problem is dealing with how to deal with multiple pointers. First it looks for toMove number from the array, if found then it looks for next non toMove variable. Once it found, it swaps these two numbers and move first and last pointer to the next number of first and last pointer.

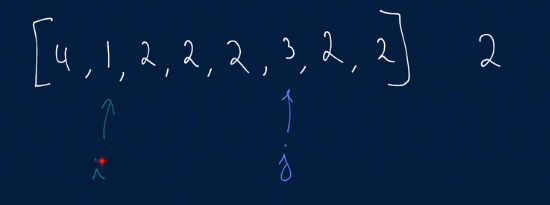


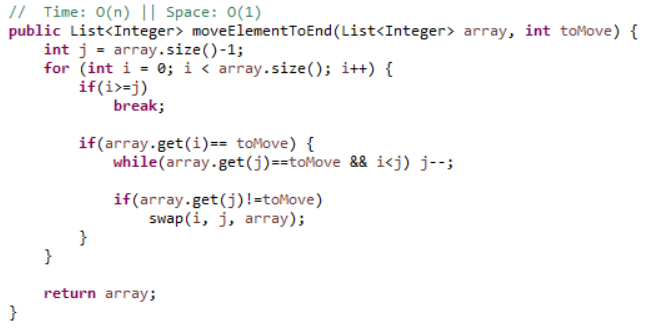
****

**Solution 2**

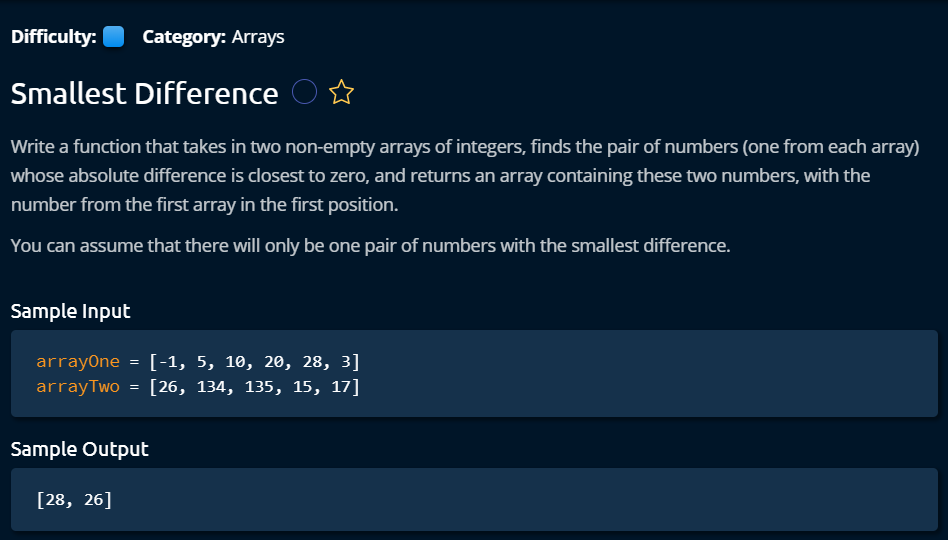
This solution is a bit elegant. It works with two sided pointers within the same loop. If left side got toMove variable, then it looks for a non toMove variable on the right side. And if it finds, then it swaps.

** **

****

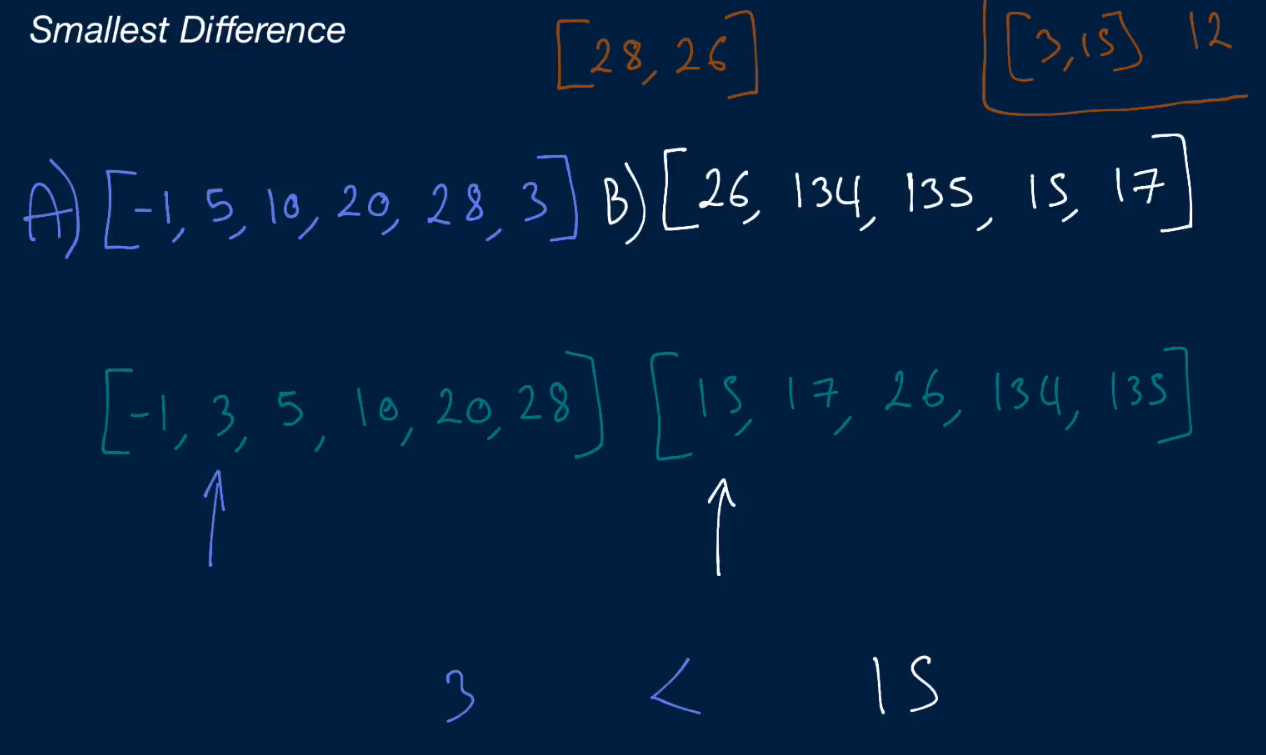
****

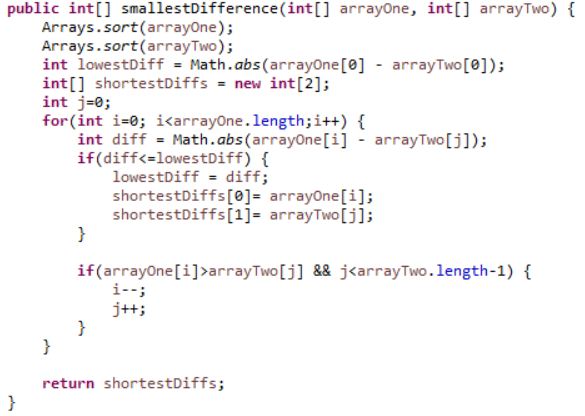
**Smallest Difference**

****

**Solution 1**

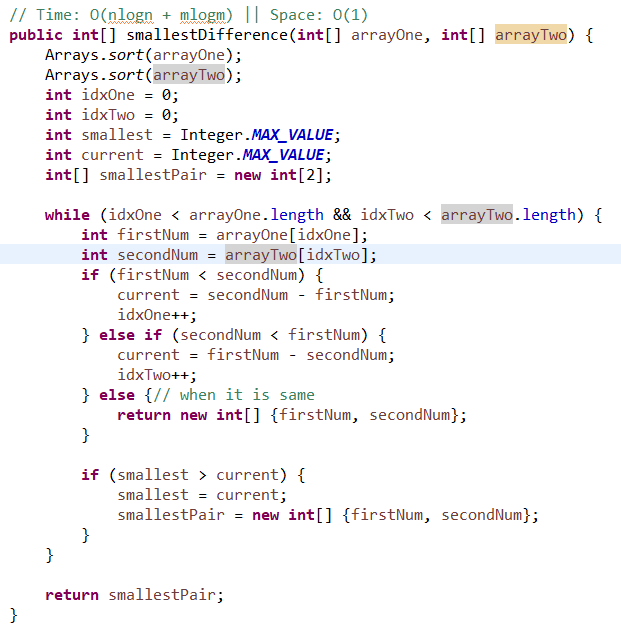
This problem is solved with two pointer management solution. It puts to pointer on two arrays and compares these two nodes value. If first value is smaller, then move that pointer to forward. If last pointer value is bigger then move it to the next pointer. We have to keep track of minimal difference of all the operation. That’s it.

****

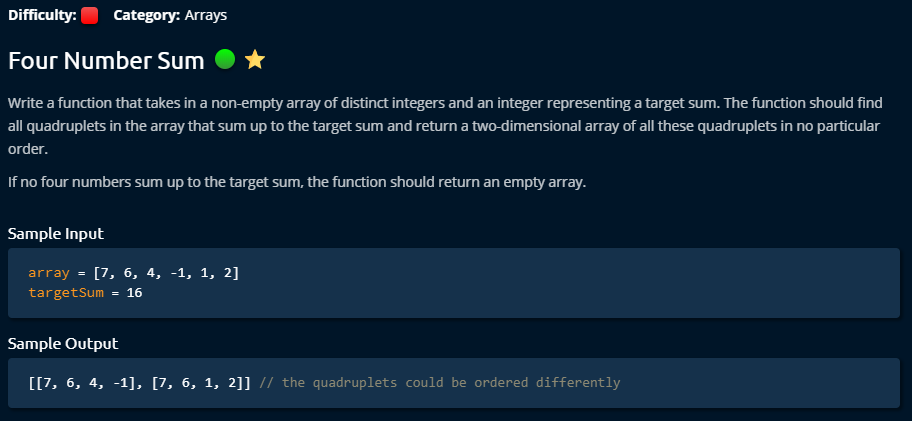
****

**Solution 2**

The solution is same but the development policy is a bit different. This kind of problem can be handled with a while loop in an even easy way. I need to adapt this while loop idea.

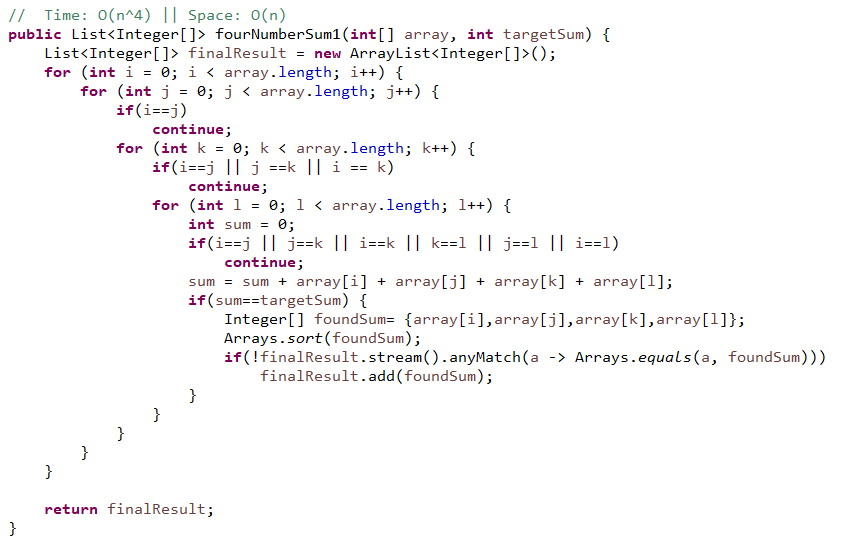
****

**Four Number Sum**

****

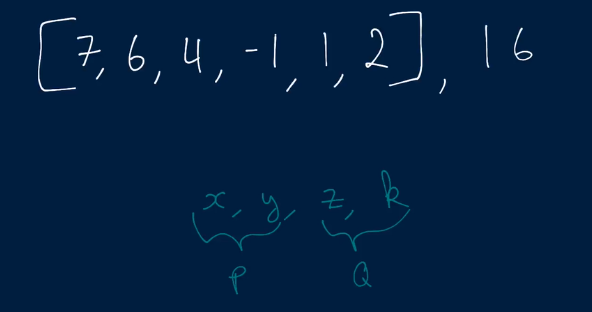
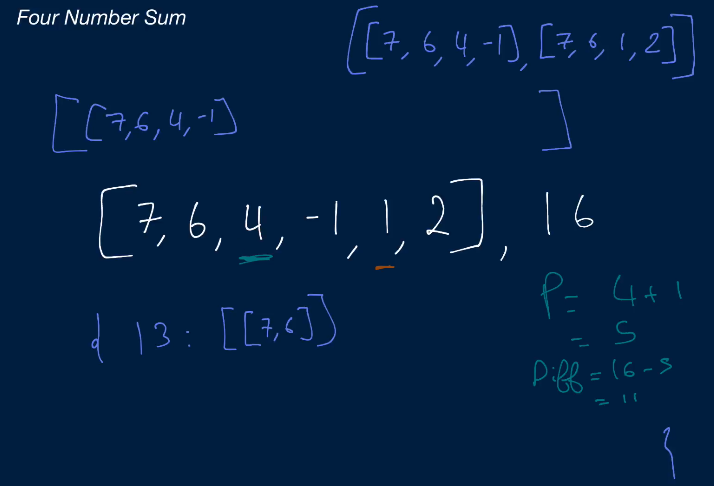
**Solution 1**

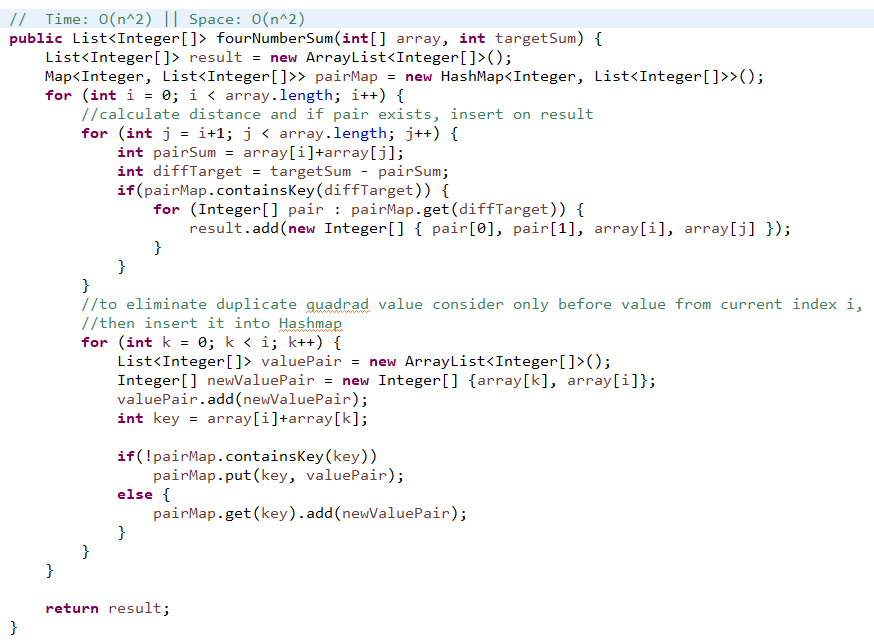
The easiest way to solve this problem by using four nested loops. The only thing is to confirm that we are not considering same index more than one places. We can do that by using i==j continue techniques. But the problem with this technique is that it takes very high time complexity (n^4) and the space complexity is good.

****

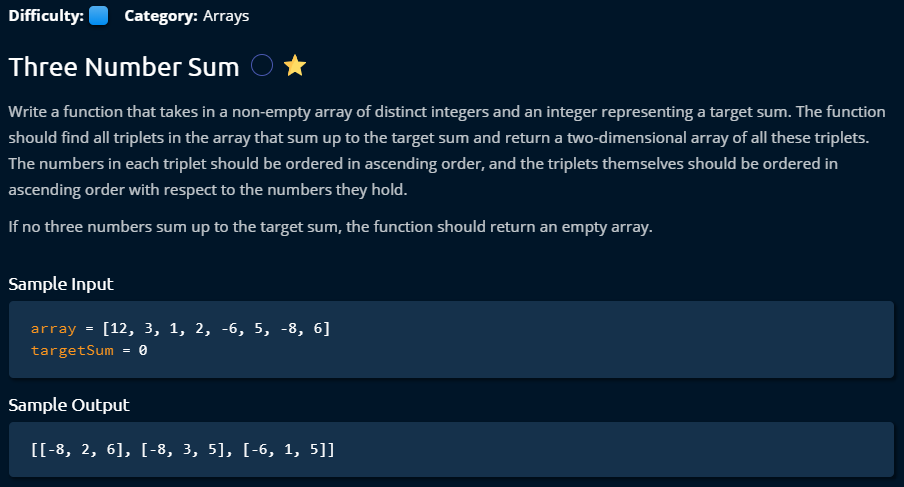
**Solution 2**

This is a bit engineered solution. This is totally new idea to me. Just by two for loop, the whole job is accomplished. To find the four numbers sum, and make the job easy, it makes a pair of two numbers and stores them on Hashmap using their sum result as a key. **Very interestingly**, to avoid duplicate data (same data with another order) in Hashmap data, it stores the key value pair into the Hashmap for the FOR loop only from the left indexes until to the current index. And the right indexes only used to find new Quadratic. It compares each pair (i, j) + Hashmaped stored pair with TargetSum, and decide found quadratic if they are same and then store it to the final list. This technique manages a surprising time complexity benefit (O (n^2)) but requires a bit more space.



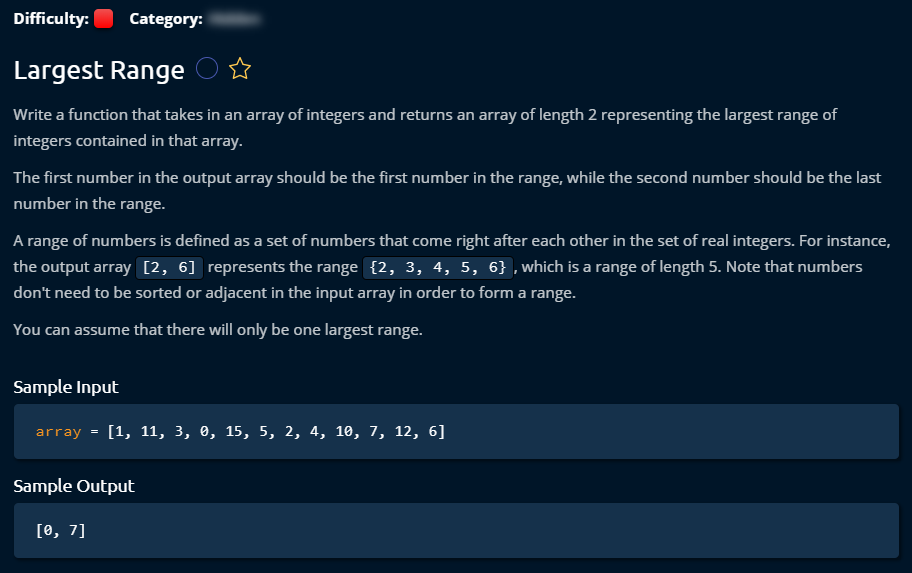
****

**Three Numbers Sum**

****

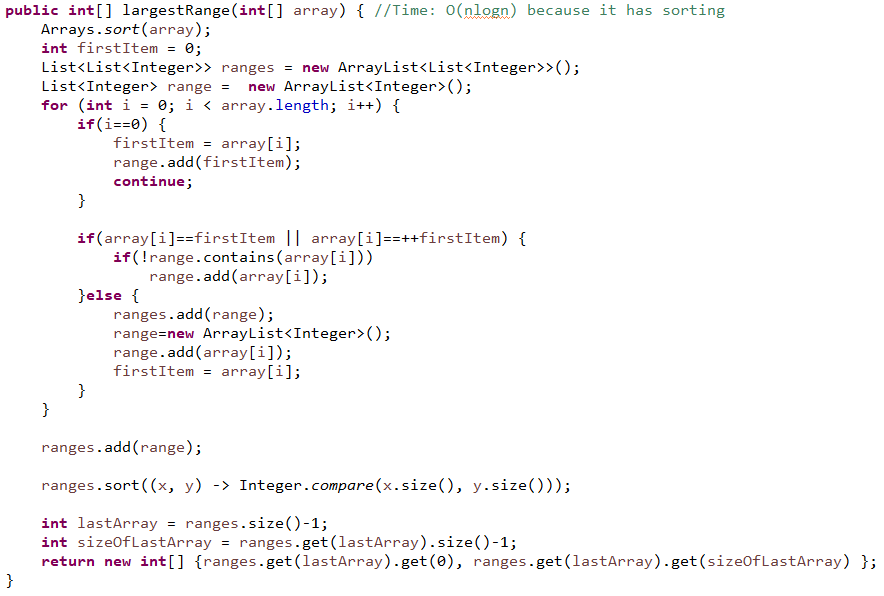
**Solution1**

**Largest Range**



**Solution 1**

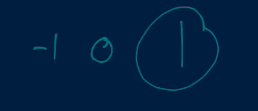
This is comparatively easy solution. But as it requires sorting, it requires O (nlogn) time and O (n) space. We can have lot more better solution by using Hashtable. We **can get O (n)** time solution.

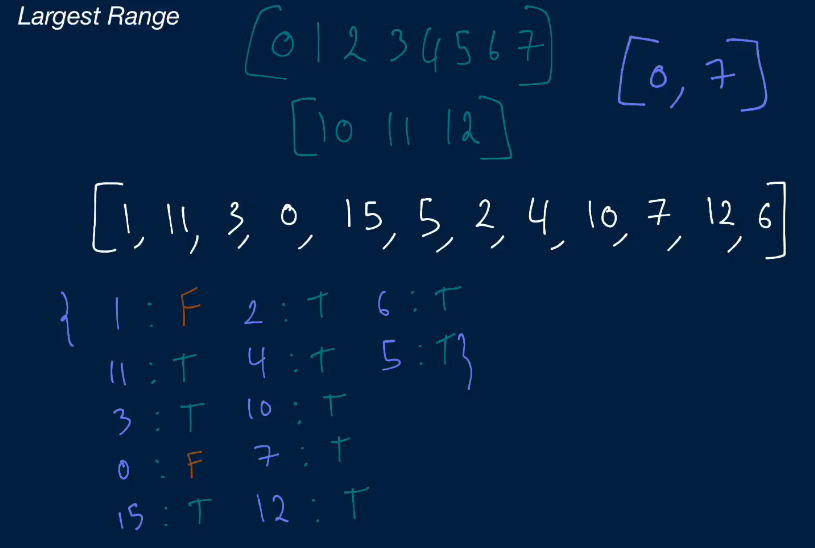


**Solution 2**

Instead of sorting, we keep these unsorted data in Hashtable where number can be a key and value can be a Boolean which can help us to keep track of if it is visited. This way we can keep it like maximum one visit of each element. Now, we have to pick one element from the input array and reduce one value and compare if exists in the hashtable. Same way increase one value and check if exists in the hashtable. We don’t have to even store the series in any array’s list. We can directly keep track of the index of biggest series and print the first and last item from them. That will reduce space uses.

Is the left number (that means -1) is in the hashtable? Is the right number (that means +1) is in the hashtable?

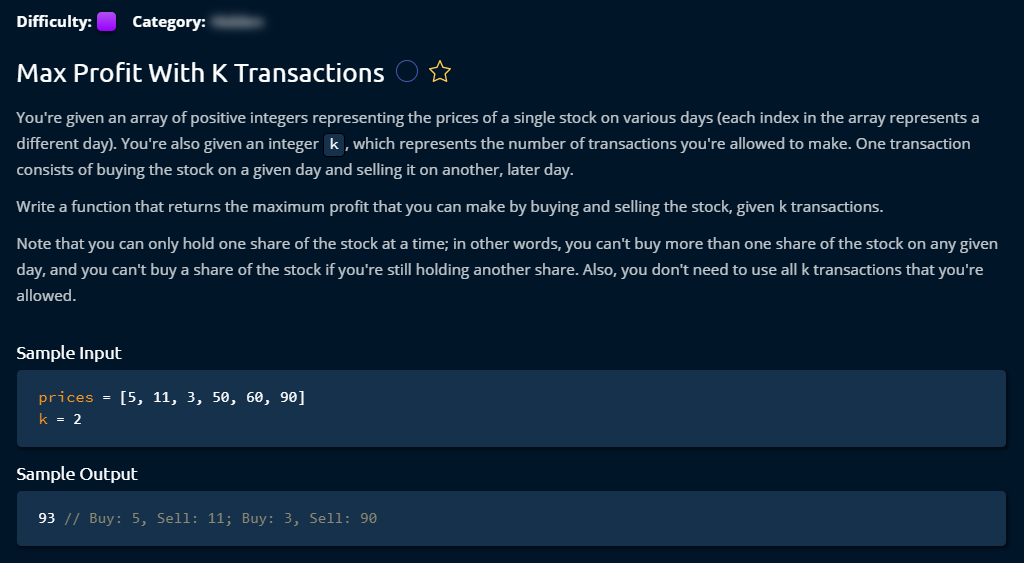


This is a genius solution with very few time complexity and space complexity. Hashmap makes it automatic sorting even we don’t use any benefit of it. One thing we have to be careful that Hashmap will remove redundant key and always use last updated key values.



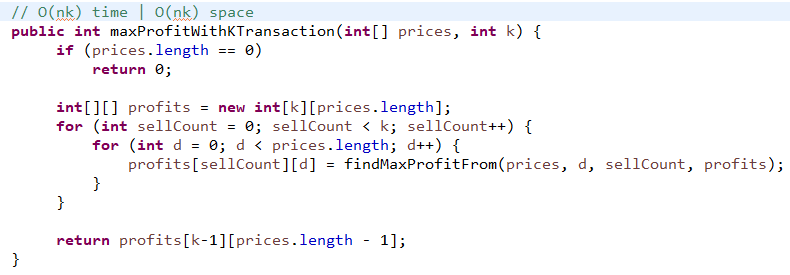
**Dynamic Programming**

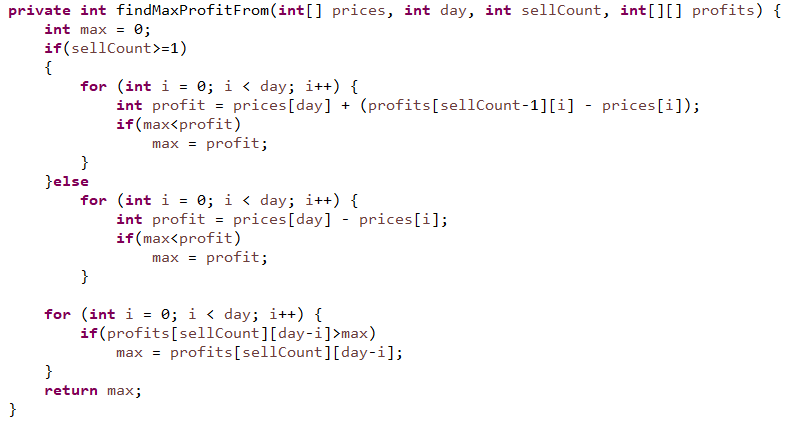
**Max profit with K Transactions**



**Solution 1**

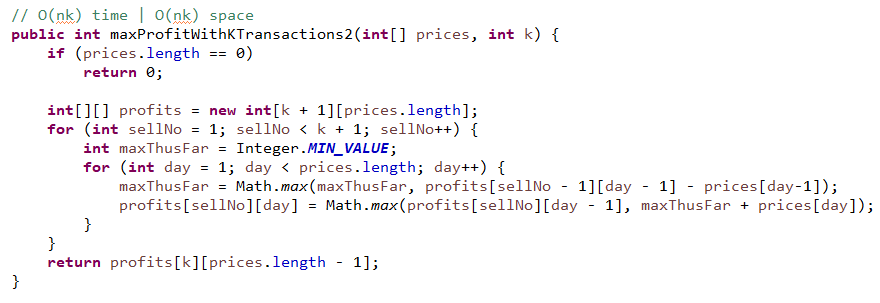
This was my solution for this problem. Probably the code implementation was lengthy. I used some of the theory from him for the calculation. This problem was a bit hard to cope up. I think I need to get back again after some more problem solving and see this one again if I feel any difference to understanding this problem.





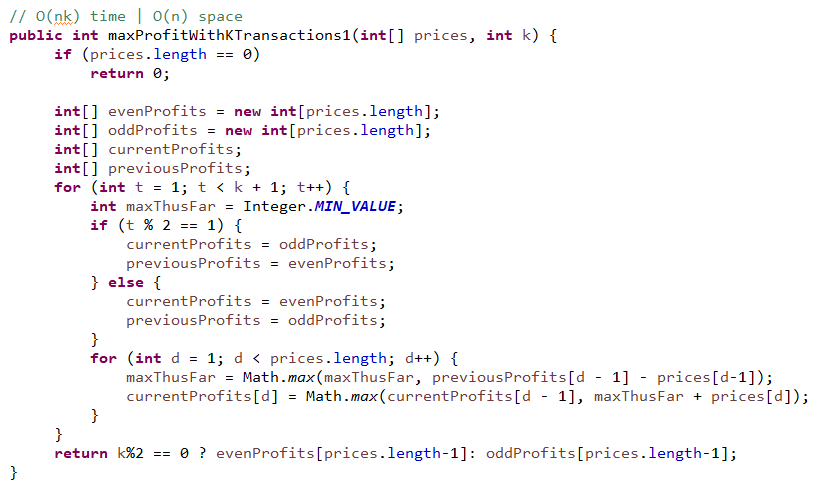
**Solution 2**

This is almost same algorithm as Solution 1. I did not properly understand how this Math.max and the discovered rules were developed. I need to come again after some day to check it.

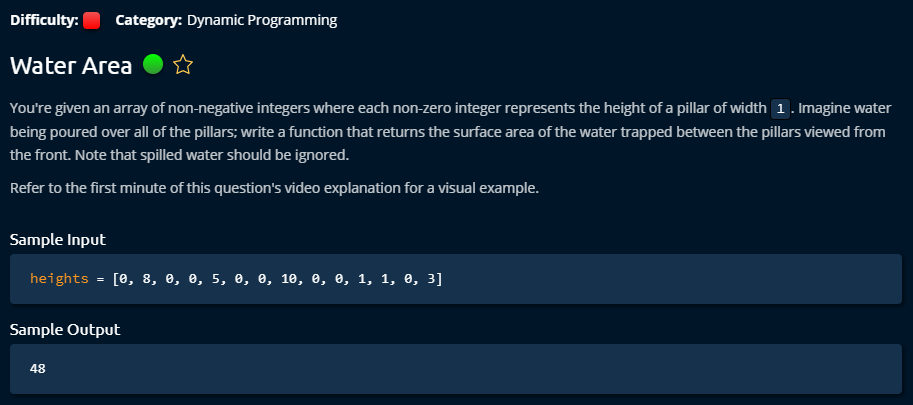
****

**Solution 3**

I did not properly understood how it reduced the space complexity

****

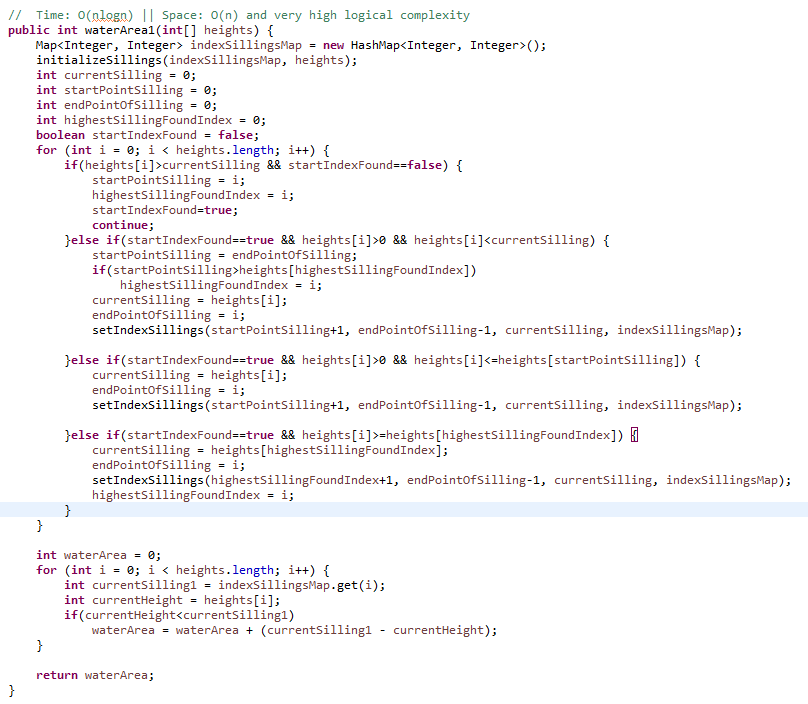
**Water Area**

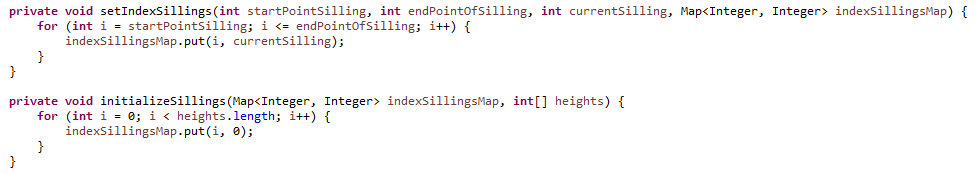
****

**Solution 1**

This one was my solution. I tried to generate sillingMap using complex logic. It worked but it turned out very hard to manage at the end. I had to manage four different conditions to store silling information. This turned out a nasty solution.

Too many if condition tells you that it is bad solution. When it is hard to manage you have to understand that you did not get right solution. And with pen and paper, I did not get the solution completely working, I just had feelings that it will work but still I have to figure out some part during the development. And it was hard.

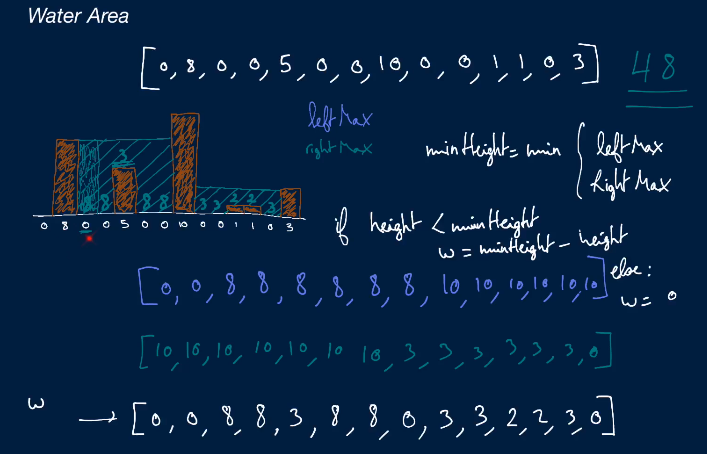
****

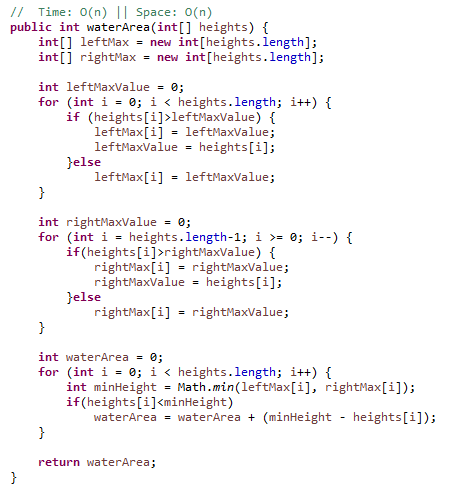
****

**Solution 2**

This is a decent solution. The theory was very decent which made the hard problem super easy to manage.

At the same time, it reduces some Time complexity. I had to generate two array which remembers left max and right max from current value of given array. And finally just decide the min height of each node, then subtracting, finally sum it up the results. The time complexity reduced to O (n) and gives a surprising code manage complexity.



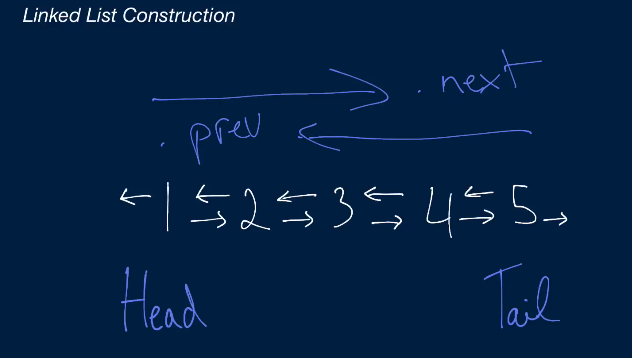
****

**Linked List**

**Construction Linked List**

****

The given Linked list to test all the functionality is following. There are two types of Linked list, Single Linked List and Double Linked List. Here is a double link list given where each node has four different connections. To manage any operation like add a node or delete a node from the linked list from double linked list, we have to deal with all the four pointers that makes linked list so interesting. During any operation of linked list, we have to consider that the list must stay stable after any operation which means it should have stable connections with other nodes.

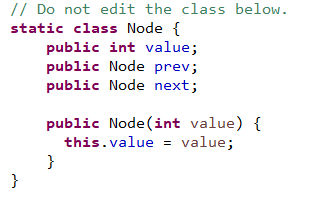
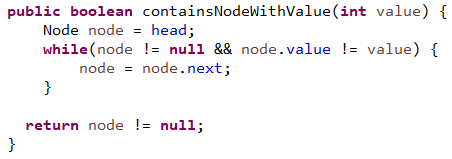
****

**Solution:**

Since we can reuse some method implementation, we can start from searching a node with a value. A class node will be given which contains node’s value and a link to next node and previous node.

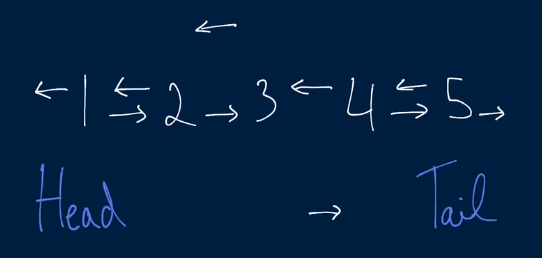
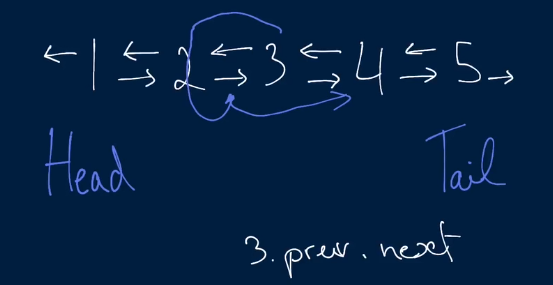
**ContainsNodeWithValue (int value)**

For searching a node from the given linked list, the first things that we have to do is, set the current pointer to the head of the given list. Then we can travel all the nodes by node.next pointer movement. During the travel if the node is null, that means we reached to the end of the list.

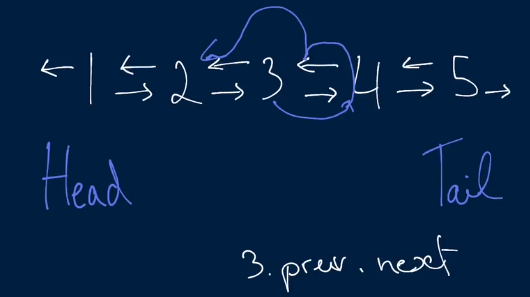
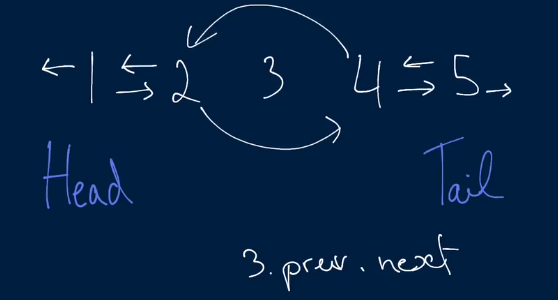
** **

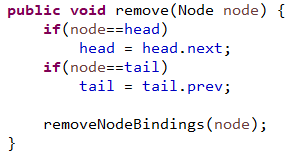
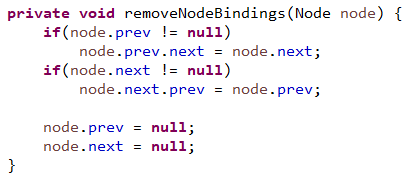
**Remove (Node node)**

To remove node, first we consider the corner cases that if the given node to delete is head or tail, then we have to move the head or tail to one node after or before. After that we can remove all of the connections from that node to other nodes.

** **

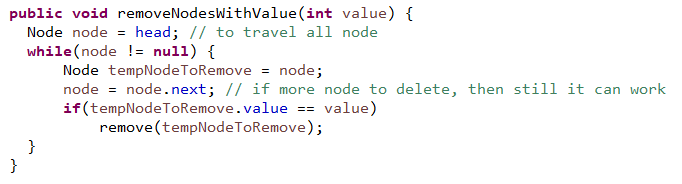
To remove the connection, first we need to think about the connections which are coming from other two nodes to the candidate node. Using Node.prev.next and Node.next.prev we can set up next and previous node’s connections to each other. Then we can set another two connections (two connections from candidate node) to null. That’s it.

** **

** **

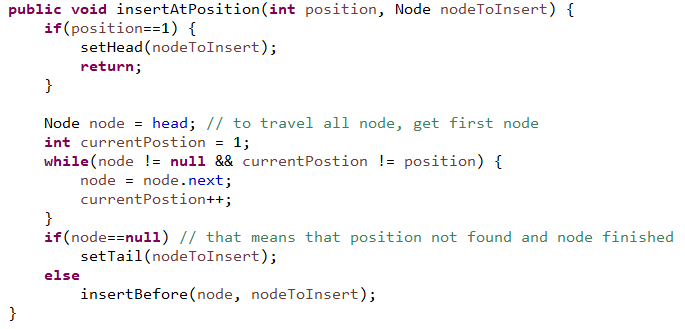
**RemoveNodesWithValue(int value)**

To remove node with particular value, first we have to search it from the list. To search, we need to set Head to the current node and using Node.next we can travel all over the nodes. By comparing by value when we find it, we can remove it using Remove method.

****

**insertAtPosition(int position, Node nodeToInsert)**

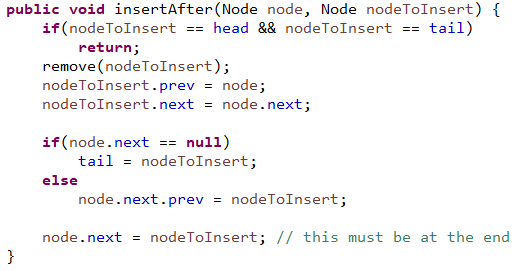
For inserting to a position of the linked list, first we need to find the node at where we to insert this node. If the position is 1, we can set this candidate to head. Otherwise, we have to travel over the node and by keeping track the number of nodes by a variable. When we find that node’s position in the list, we insert it by just using the method InsertBefore of that node. If we don’t find it, the last node will be null, so need to insert this candidate Node to tail of this node.

****

**insertAfter(Node node, Node nodeToInsert)**

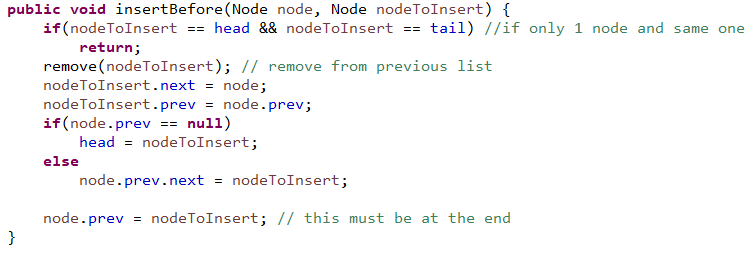
Anode has to be inserted where first we need to remove that node from the given list. First of all, we need to think about the corner case that if there is only one node came in the given list and if the node is same as the candidate node, then we do nothing and return because it makes no sense to remove and insert same node.

Other wise after removing the node, we need to take care all the four pointers again for the new node. As we have the information, first we insert prev and next node for the candidate node. Then we need to think about if the next node is exists, if yes then the link from previous node to the candidate should be set. And finally we can set the given node’s next pointer to the candidate node. That is it.

****

**insertBefore(Node node, Node nodeToInsert)**

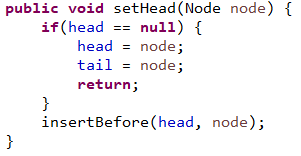
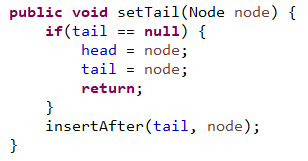
InsertBefore is very similar to insertafter method, only the node’s pointer to take care is left oriented.

****

**setHead(Node node), setTail(Node node)**

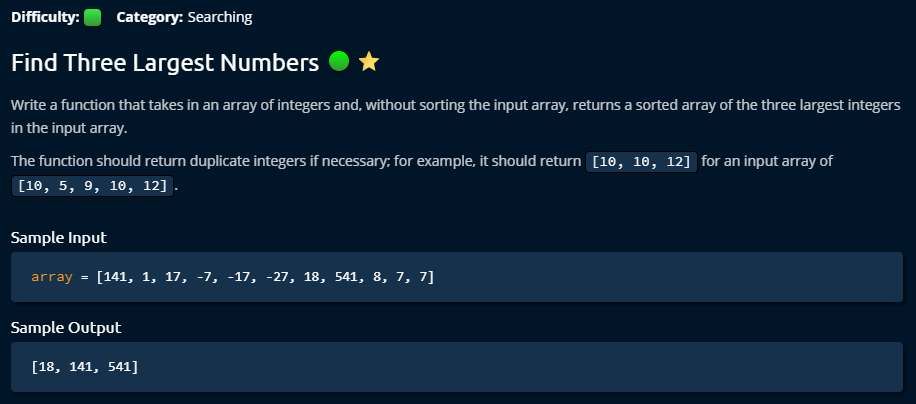
To set Head and Tail, we need to use some of the existing method. Both of them are very similar. First we need to take care the corner cases as usual. For the case SetHead (node), if the head is null that means it is an empty list, then we only do set head and tail to the candidate node and return. Otherwise we just insert this candidate node before existing head. That’s it.

For the case of SetTail, we need to insert the candidate node after the tail node. That’s it.

** **

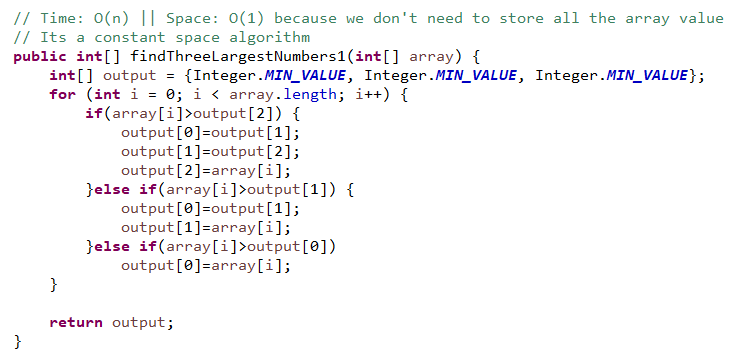
**Searching**

**Find Three Largest Numbers**

****

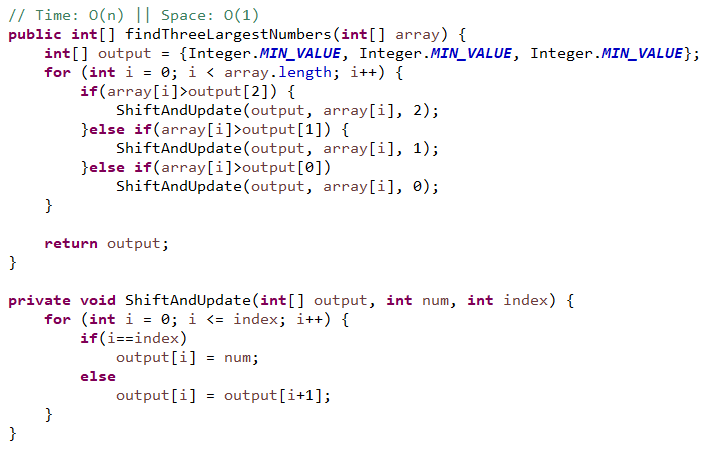
**Solution 1**

This solution came from me. I am happy that I was able to do it on 7th day. And the time and space complexity remain elegant from my solution.

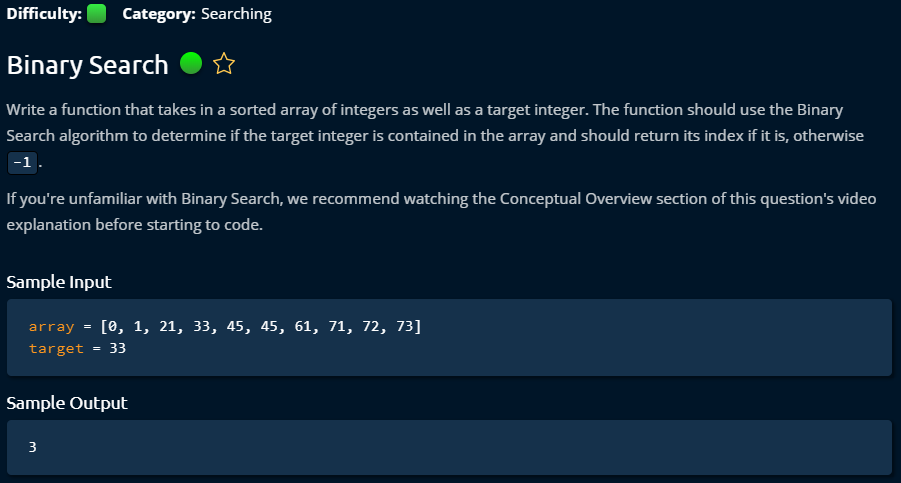
****

**Solution 2**

This is quite similar like my solution. The only difference is that it extracts a method to do dynamically the operations. And the time and space complexity remain same.

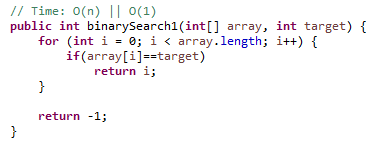


**Binary Search**



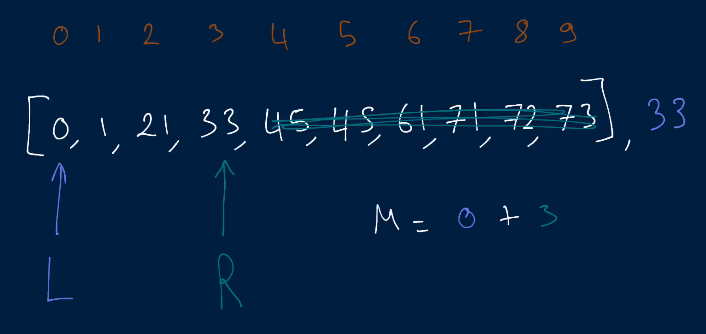
**Solution 1**

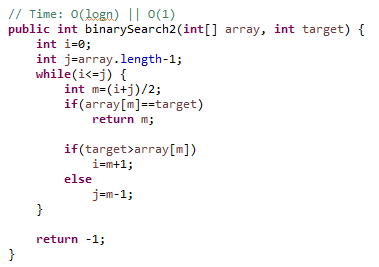
To do binary search, input array must be sorted. My general solution which costs O(n) in worst case when data not exists.

****

**Solution 2**

This is an elegant solution which reduces a significant amount of workload by eliminating half of data in each iteration. It starts with two pointers in left and right side of the array. Then it calculated middle index, and if the middle index is more than the target value, then left index is changed to middle +1. That’s how it reducing half of it every time.

****

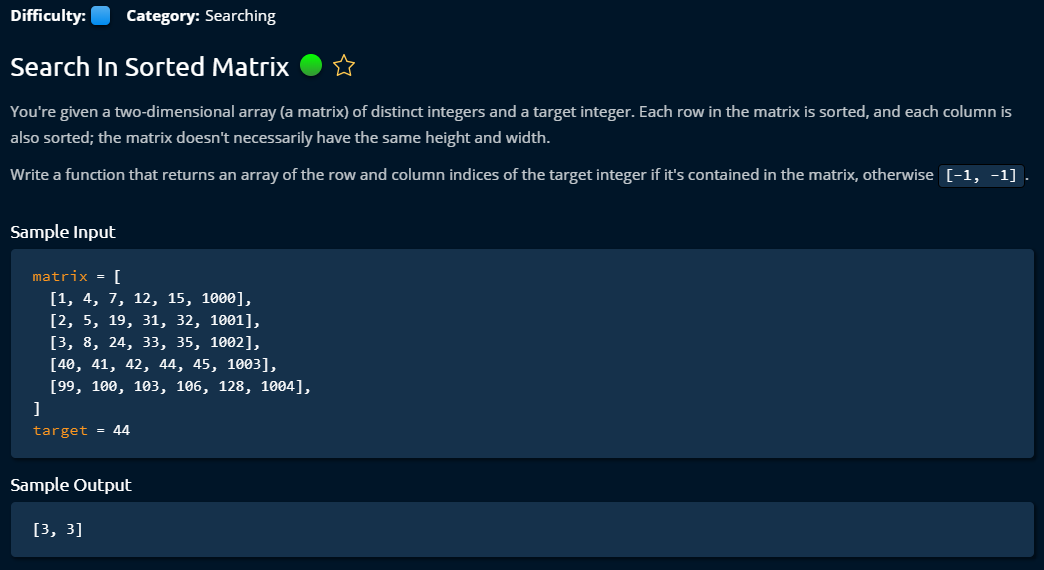
****

**Solution 3**

The same while loop job done in Solution 2 is done here by a recursive method calls. It shows recursive method job can be done in while loop. I like while loop more because to do recursive call we have use call stack which is not memory friendly (it can make problem when you have limited memory in server) and hard to track bugs, but while loop don’t need that.

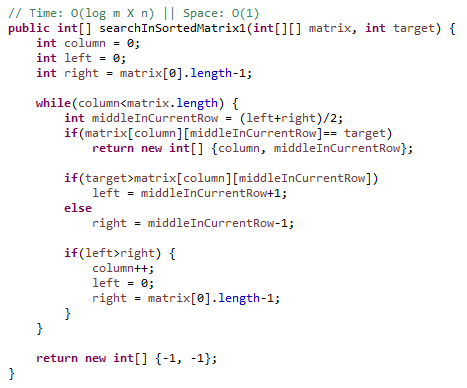
****

**Search in Sorted Matrix**

****

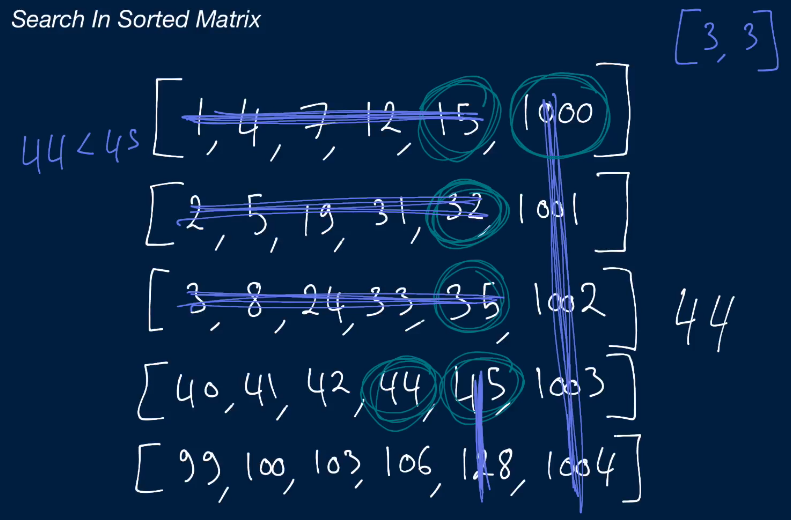
**Solution 1**

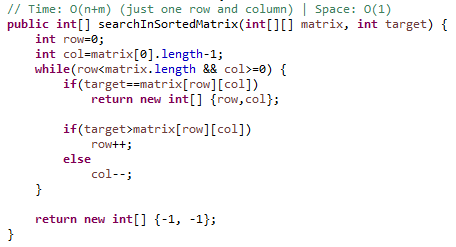
It was my solution. I used same idea of binary search in each row, when it is not found then moved to next row.

****

**Solution 2**

This is a genius solution from Clemen. It can be used many places. It starts with first row last index. If target is bigger than current index, then we move down by increasing row, otherwise we move left by reducing column. That’s we have to just visit maximum one row and one column to find out if target is there.



****

**Binary Search Trees**

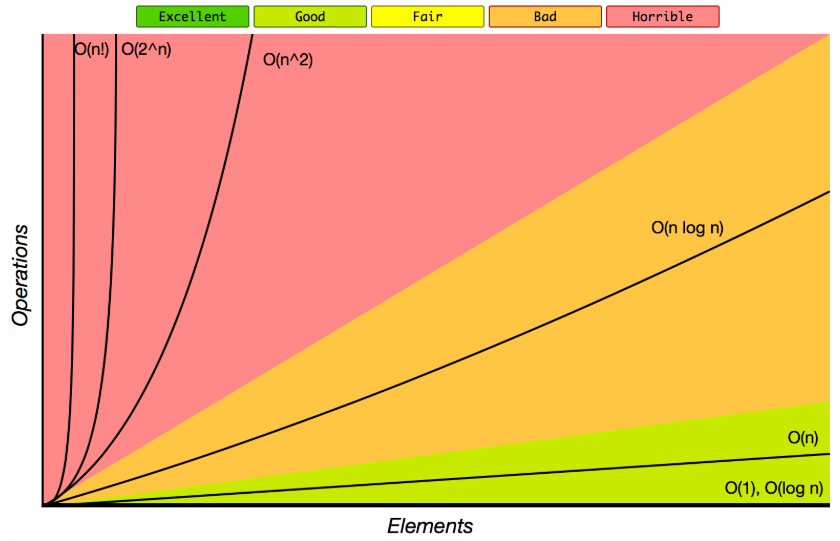
**BST Construction**

****

**Solution**

This was very complex to manage. I lose interest to make its document. I will make its document later when I am finished with everything else.

**O (N) Chart**



**Top 10 Algorithms for the Coding Interview,**

1. Depth First Search

2. Breadth-First Search

3. Matching Parenthesis

4. Making use of Hash Tables

5. Knowing how to manipulate multiple variables/Pointers at once (p1)

6. Reversing a linked list

7. Sorting fundamentals (time complexity, functioning, etc.)

8. Recursion

9. Custom Data structure (ex. suffix tree)

10. Binary Search (p1)

Here's the list if you want to search the topic's, still TechLead just nails it explaining each one on the video.

1. Solving the problem itself is not the goal, it is really the process that means the analysis of the problem, how well you can analyze it, and how well can you look at the alternatives considering pros and cons in terms of time and memory and bigO time analysis. You should be able to propose something like if you want optimize time complexity we might use this data structure and solve it this way, but if you want to optimize the memory we need to do it this other way.
2. Tree traversal: is used heavily in industry, how to traverse a tree: preorder traversal, in order traversal, post order traversal, breath first search and depth first search. One common problem is, given a hierarchy where there are views, sub views, sub-sub views: traverse every view and print every view, you should not do it recursively and iteratively, use breath first search to find the nearest node that match the criteria
3. Binary search tree: you need to traverse some tree structure looking for some node, that can be done in logarithmic time, you have to know the data structure to represent a tree. In the data structure course, they teach you it as just having a node with left node and right node, this is kind of naïve because that limit the fanout to two. If you want more larger fanouts, you can use an array, an array of children nodes. Recursions are the common way to traverse this tree structures.
4. Recursive function: is tricky to properly understand, most people do not know it elegantly. Commonly happens that when you need to write a recursive function, you need to pass parameters to pass the state, then recursive function gets bigger and bigger and you need to pass a bunch of variable into it. It is better to have helper function in recursive function where you call into the recursive program and the recursive program call straight into the helper function but passes in a set of initialization variables that keeps your code API clean. You have to think about what your base case is? If you can clarify under what condition it will end, that helps you to write cleaner recursive functions. Recursion is limited to stack space, because it has limits, it is not that optimal. But what professional programmer mostly does is, convert it to iterative function.
5. Iterative algorithm: Every recursive function can be converted to an iterative algorithm using stack and queues. Try implementing Fibonacci and Tree traversal iteratively. Some recursive function is pretty difficult to convert do it iteratively, it depends on how much state you are passing from one recursive to the next one.
6. Stacks and queues: Very common: number of parenthesis problem, you can do it using stack.
7. Object oriented programming: You know the code structure, making class for anything concise.
8. Data structures (Hashmaps, graphs, etc.)

Less but still important:

1. Pointers in the program how to control them efficiently
2. Sorting algorithms (Merge sort, quick sort, bubble sort)
3. Strings and how to navigate through them (Some types: palindrome, anagram)
4. Dynamic programming: Many times it is involved memorization or caching the values and reuse them.