**Java Trick and Techniques to make better**

**Two ways we can save updated value to a variable,**

maxSum = Math.*max*(sum, maxSum); //cleaner option or

if(maxSum>sum)

maxSum = sum;

**Difference between boolean primitive and Boolean object initialization,**

**boolean**[][] isVisited = **new** **boolean**[array[0].length][array.length];//by default all value will be false

**Boolean**[][] isVisited = **new** **Boolean**[array[0].length][array.length];//by default all value will be null

**It is possible to null in the method parameter for normal or recursive method and later in the method declaration you can declare the Object type. (Smart way of call method)**

**public** **static** BST minHeightBst(List<Integer> array) {

**return** *getMinHeightBST*(array, **null**, 0, array.size()-1);

}

**private** **static** BST getMinHeightBST(List<Integer> array, BST bst, **int** startIdx, **int** endIdx) {

**return** **null**;

}

**Pre-order, In-order, Post-order tree traverse by recursive call,**

By two recursive methods call (with left and right node), it is possible to traverse any tree by three way (pre, in, post order traverse). Now these three types of traverse can be manipulated by the order of data collection location.

When it is before of two recursive call, that’s pre order traverse,

array.add(tree.value);

*preOrderTraverse*(tree.left, array);

*preOrderTraverse*(tree.right, array);

When it is middle of two recursive call, that’s in order traverse,

*inOrderTraverse*(tree.left, array);

array.add(tree.value);

*inOrderTraverse*(tree.right, array);

When it is after of two recursive call, that’s post order traverse,

*postOrderTraverse*(tree.left, array);

*postOrderTraverse*(tree.right, array);

array.add(tree.value);

**We can reverse a list of data by stack,**

**while**(!stack2.isEmpty()) { // stack2 : 10,15,22,5,5,2,1

array.add(stack2.pop()); // array: 1,2,5,5,22,15,10

}

**How to manage two dynamic variable recursively?**

**private** **static** **boolean** validateBST(BST tree, **int** minValue, **int** maxValue) {

**if**(tree.value<minValue || tree.value>=maxValue)

**return** **false**;

**if**(tree.left != **null** && !*validateBST*(tree.left, minValue, tree.value))

**return** **false**;

**if**(tree.right != **null** && !*validateBST*(tree.right, tree.value, maxValue))

**return** **false**;

returen true;

}

Recursive is very similar like stack. It puts number one top of each other. And first works on top one.

How to understand recursive method easily, Need to try and make documents,

<https://www.youtube.com/watch?v=B3U6LExgevE>

<https://www.youtube.com/watch?v=BibDrTCGXRM>

<https://www.youtube.com/watch?v=oKndim5-G94>

<https://www.youtube.com/watch?v=lMBVwYrmFZQ>

<https://www.youtube.com/watch?v=6oDQaB2one8>

<https://www.byte-by-byte.com/recursion/>

Recursive method is calling using stack behind. It is same like when we add node in DFS stack and pop last inserted one node when base case came. Process it and then pop another node from stack. So, recursive method we can debug or analyze using stack paper debug or drawing tree and drawing its each node’s return.

Recursive is very similar like a loop. We just have to provide that parameters and base case condition which should be opposite way written in compare to while loop in the previous solution. That is it. **One recursive called method** is very similar like **a while** method. **Two recursive methods** must be described by **tree structure.** In recursive method, we have to reduce the input array/set at each iteration, so that over time we can reach to base case. That is way how it works. Even if in the base case has no exclusive return written (if something for example added in the list), it will go out if there is only if else condition there.

**When you feel you need unlimited (many) for loops to solve any problem, that is a symptom that this problem is a classic recursive problem. Permutation or Knapsack problem is a great example for that**

**There are 6 patterns** (based on basic recursive principles) of recursive methods,

1. **Iterative:** 
   1. Iterative over an array/list using recursion
   2. Rarely useful except for simplifying code
   3. Example: Factorial, Print linked list reverse order, any loop use
2. **Breaking into Sub problems**
   1. Classic recursive problems, how can we reduce my input by size 1, and ultimately get to the solution
   2. Use this patterns when it make sense to you
   3. Example: Tower of Hanoi, Fibonacci
3. **Selection (combinations)**
   1. Fundamentally, the problems that can be solved by finding all valid combinations.
   2. Brute force – find and validate every combination
   3. Backtracking or optimize by dynamic programming
   4. Example: Knapsack problem, NQueens, Phonespell, Word break
4. **Ordering (permutation)**
   1. Similar to selection except orders are important here
   2. Brute force – find all permutation and validate which one is best matches for our conditions
   3. Example: Permutation, Find N digits numbers whose sum to a specific value, Word squares
5. **Divide and Conquer**
   1. Classic recursive problems, similar to breaking into sub problems, but here is splitting problem into two half
   2. Common with searching, sorting, tree
   3. Example: Mergesort, Find all valid parenthesis, Generate BSTs for a set of items
6. **Depth first Search**
   1. Common techniques for tree and graphs structures
   2. Can be used for many other different recursive problems
   3. Examples: Search in tree, Probability of a knight on a chessboard

**Boolean ? Operator can be provided on if or while condition too,**

**if**(isLeftSide? curNode.left.value>=max : curNode.left.value<=max)

or,

if(isLeftSide? True : false)

**Java 10 you can write var variable name,**

**var** alphabet = "your\_name";

**How to get smallest number from the List<Integer> using java stream?**

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

list.stream().min(Comparator.*comparing*(Integer::*valueOf*)).get(); // smallest integer

String minChar = Stream.of("H", "T", "D", "I", "J").min(Comparator.comparing(String::valueOf)).get();

**How to check if a particular letter is part of any particular signs String?**

String openBrackets = "({[";

**if**(openBrackets.indexOf(sign)!=-1)

//That means sign exists here

**How to map related data which can be shortly comparative?**

Map<Character, Character> map = **new** HashMap<Character, Character>();

map.put(')', '(');

map.put('}', '{');

map.put(']', '[');

**How to sort all letter from a word?**

**String word = “act”**

**char** wordArray[] = word.toCharArray();

Arrays.*sort*(wordArray);

String sortedWord = **new** String(wordArray);

**How to write test case for List<List<String>>? And how to know if any particular list contains there when the output is not ordered? How to initialize new ArrayList simply?**

@Test

**public** **void** test() {

GroupAnagrams obj = **new** GroupAnagrams();

List<List<String>> results = obj.groupAnagrams(**new** ArrayList<String>(

Arrays.*asList*("yo", "act", "flop", "tac", "foo", "cat", "oy", "olfp")));

*assertTrue*(contains(results, **new** ArrayList<String>(Arrays.*asList*("yo", "oy"))));

*assertTrue*(contains(results, **new** ArrayList<String>(Arrays.*asList*("flop", "olfp"))));

*assertTrue*(contains(results, **new** ArrayList<String>(Arrays.*asList*("act", "tac", "cat"))));

*assertTrue*(contains(results, **new** ArrayList<String>(Arrays.*asList*("foo"))));

}

**public** **boolean** contains(List<List<String>> arr1, List<String> arr2) {

**for** (List<String> subArray : arr1) {

**if** (subArray.equals(arr2)) {

**return** **true**;

}

}

**return** **false**;

}

**How to find a particular match of substring?**

One approach is stay center; compare left and right by two pointers. For example, LongestPalindromeSubstring, we need to find the center of a palindrome from long string.

String biggestPalindrome = Character.*toString*(str.charAt(0)); // take the first letter as a longest palindrome,

**for** (**int** i = 1; i < str.length(); i++) { //so it does not bother at comparing left and right

**if**(str.charAt(i-1)==str.charAt(i)){}

**if**(**i<str.length()**-1 && str.charAt(i-1)==str.charAt(i+1)){} // at only last letter, it will not compare

// so **no out of bound exception**

**How to make all possible substring of a string?**

String str="abbc";

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

**for** (**int** j = i+1; j <= str.length(); j++)

String subStr = str.substring(i,j); //a, ab, abb, abbc, b, bb, bbc, b, bc, c

**What is backtracking algorithm in data structure?**

**Backtracking** can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem. Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time. E.g. Dynamic programming

**What is difference between greedy algorithm and dynamic programming?**

In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution. Such algorithms are called greedy because while the optimal solution to each smaller instance will provide an immediate output In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution.

**Comparing something in between a given unsorted Array?**

To make thing faster, always think first if you can do something with taking a two length array (new int [2]) or think about Hashtable with Boolean (it will remember if you already accessed a value from the array). You can even use Boolean two dimensional array (Boolean[ ][ ] name).

**Sorted array is prerequisite** for binary search. If a given array is sorted, then it more likely to use binary search there.

**How to get always positive difference?**

int curDist = Math.abs(12 - 15); // 3 not -3

**How to get max number from two numbers?**

int maxNumber = Math.max (12, 15); // 15 int minNumber = Math.min (12, 15); // 12

**Convert String to character array.**

**char**[] characters = smallString.toCharArray();

**How to fill up an array with maximum number?**

**int**[] jumps = **new** **int**[array.length];

Arrays.*fill*(jumps, Integer.***MAX\_VALUE***);

**To compare and manipulate a string, we must not split it into characters; we can directly use it in the loop which could save a complete n space.**

String str=”abcdeabcdefc”;

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

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

**How to generate list of list using number to list directly? For example, [10, [1,2]]**

List<List<Integer>> result = new ArrayList<List<Integer>>();

List<Integer> totalValue = Arrays.asList(10);

List<Integer> finalItems = Arrays.asList(1, 2);

result.add(totalValue);

result.add(finalItems);

**When any algorithm containing sorting, that means it will definitely minimum?**

**O (nlogn): time and O (n) space**, Java use by default **Timsort** which is combination of Mergesort and Insertionsort. Some sorting like insertion, bubble, selection has O (1) space complexity but O (n^2) time complexity. Quicksort has best, average case O(nlogn) time. Still worse case has O (n^2) time. But Mergesort has all case O (nlogn): time and O (n) space**.**

**Best way to initialize List in Java 8,**

List<Integer> sortedArray = **new** ArrayList<Integer>() {{

add(1);

add(2);

add(3);

add(5);

add(6);

add(7);

}};

**Java how to convert list to array using java 8,**

sortedArray.stream().mapToInt(k->k).toArray();

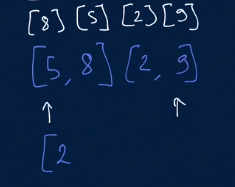
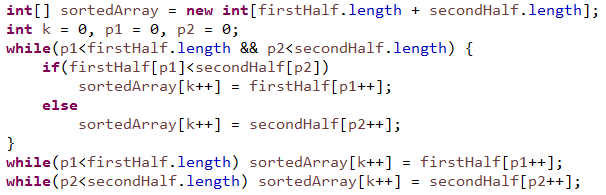
**How to separate parts of array,**

**int** middle = array.length/2;

**int**[] firstHalf = Arrays.*copyOfRange*(array, 0, middle);

**int**[] secondHalf = Arrays.*copyOfRange*(array, middle, array.length);

**How to insert value and at the same time increase the pointer to get and set next value?** It compared and insert from two sorted array into another new array.

**How to deliver pointer of multiple index with the array to do an operation, not the index value?**

swap(i, i+1, array);

**Best way to initialize array in java,**

**int**[] expected = {1,2,2,2,5};

**int**[][] matrix = {{1,0,0,1,0},

{1,0,1,0,0},

{0,0,1,0,1},

{1,0,1,1,0},

{1,0,1,1,0}};

How to convert array to list,

Arrays.toList(myArray);

**How to sort an array?**

Arrays.*sort*(array);

Arrays#Sort() for object arrays uses TimSort, which is a hybrid of **MergeSort** and InsertionSort.

**Array Sort in Reverse Order,**

Integer[] array = **new** Integer[] {};

Arrays.*sort*(array, Collections.*reverseOrder*());

**How to check if an array contains any particular element,**

Integer[] intArray = new int[]{2,4,5,7};

**if**(Arrays.*asList*(intArray).contains(5) {}

**How to check if any List of arrays contains any particular Array,**

List<Integer[]> finalResult = **new** ArrayList<Integer[]>();

Integer[] testArray= {1,2,3,4};

Boolean found = finalResult.stream().anyMatch(a -> Arrays.*equals*(a, testArray));

**For list of list, it is enough to compare with list contains,**

List<List<Integer>> powersets = **new** ArrayList<List<Integer>>();

If(powersets.contains(array))

powersets.add(array);

**How to test/compare a List of arrays using unit test,**

ThreeNumberSums obj = **new** ThreeNumberSums();

List<Integer[]> receivedResult = obj.threeNumberSum(

**new** **int**[] {12,3,1,2,-6,5,-8,6}, 0);

*assertThat*(receivedResult.get(0)).isEqualTo(**new** Integer [] {-6, 1, 5});

*assertThat*(receivedResult.get(1)).isEqualTo(**new** Integer [] {-8, 3, 5});

*assertThat*(receivedResult.get(2)).isEqualTo(**new** Integer [] {-8, 2, 6});

**How to initialize list of Integer for test/compare?**

List<Integer> intList = **new** ArrayList<Integer>(Arrays.*asList*(1,2,3));

**How to initialize complex list of Object for test/compare?**

// [5, 2, [7, -1], 3, [6, [-13, 8], 4]]

List<Object> complexList = **new** ArrayList<Object>(

Arrays.*asList*(5,2,**new** ArrayList<Object>(Arrays.*asList*(7, -1)),3,

**new** ArrayList<Object>(Arrays.*asList*(6, **new** ArrayList<Object>(

Arrays.*asList*(-13, 8)), 4))));

**How to compare if an Object is ArrayList type? How to convert from Object item to ArrayList<Object> ?**

List<Object> complexList // [5, 2, [7, -1], 3, [6, [-13, 8], 4]]

**for** (Object item : complexList) {

**if**(item **instanceof** ArrayList) {

@SuppressWarnings("unchecked")

ArrayList<Object> innerItem = (ArrayList<Object>) item;

sum = sum + *getProductSum*(innerItem, multiplier+1);

}

}

**A very useful technique from Clemen to keep track of pairs in adjacent node or pair of Sum,**

Map<Integer, List<Integer[]>> pairMap = **new** HashMap<Integer, List<Integer[]>>();

**How to eliminate/avoid duplicate element when you want to compare from an array?**

You store element and its location to a Hashmap; later on you can compare the value and location to make sure if duplicates happen.

Map<Integer, Integer> inputMap = **new** HashMap<Integer, Integer>();

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

inputMap.put(array[i], i);

if(inputMap.containsKey(key) && inputMap.get(key) != i){}

**Insert value into the map,**

Integer[] newValuePair = **new** Integer[] {5, 7};

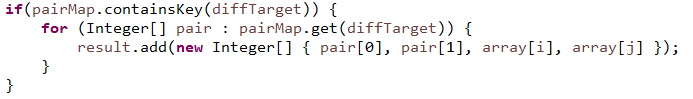
pairMap.put(12, newValuePair);

**How to add second Interger[] for that same key into that list,**

Integer[] anotherValuePair = **new** Integer[] {6, 6};

pairMap.get(12).add(anotherValuePair);

**How to get the list of Integer from that list and use it,**



**How to get map key by value?**

Map<String, Integer> charMap = **new** HashMap<String, Integer>();

String foundNewStr = *getKey*(charMap, 2);

**public** **static** <K, V> K getKey(Map<K, V> map, V value) {

**return** map.entrySet().stream()

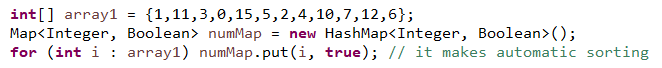
.filter(entry -> value.equals(entry.getValue()))

.map(Map.Entry::getKey)

.findFirst().get();

}

**When we need an array to be sorted, we have to remember that hashmap will remove any redundant keys. The only last updated one will be there.**



**If all Boolean data are true from a Boolean array we can use this, But contest do not allow IntStream.**



**Fastest key value pair data structure in java,**

Any hash-based [Map](http://java.sun.com/javase/6/docs/api/java/util/Map.html) structure is the way to go as long as your [hash function](http://en.wikipedia.org/wiki/Hash_function) for the key is efficient. You can use value id:s as the result for the lookup to conserve memory during search.

Map<Integer, Integer> myMap = **new** HashMap<Integer, Integer>();

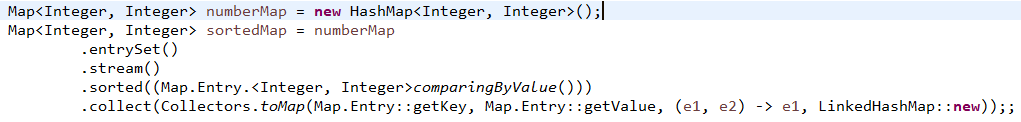
In single-threaded context, HashMap is the fastest option but it is not synchronized.

For **add, remove** method, it is even as much as 3x more efficient than others. Only **get** is only a bit faster on **ConcurrentHashMap**, but not much.

If the access to the map is multithreaded, **ConcurrentHashMap** is synchronized and is really the best solution. Hashtable is also synchronized and works for multi-threaded environment.

Map<String, Integer> myConcurrentMap = **new** ConcurrentHashMap<>();

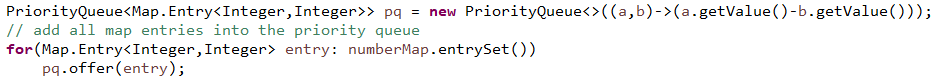
**When you need to sort a map by its value using lambda function using java 8,**



Remove key value pairs containing value 0,

sortedMap.values().removeIf(f -> f == 0);

**That same (hashmap) sorting job can be done by taking a priority queue and inserting that maps value in it, it will be lot faster as we don’t need that sorting.**

****

**This code with List of Map.Entry from LinkedList does that sorting by normal Collections.sort even faster,**

****

**How to sort list of Integers?**

List<Integer> itemIndexs = **new** ArrayList<Integer>();

Collections.*sort*(itemIndexs);

**How to add item in a particular location of a list?**

itemIndexs.add(0, i-1);

**List of list of Integer sort by first element of inner list, here x is the first list and y is another list to compare (ascending).**

List<List<Integer>> ranges = **new** ArrayList<List<Integer>>();

ranges.sort((x, y) -> Integer.*compare*(x.get(0), y.get(0)));

**List of list of Integer sort by internal list size,**

List<List<Integer>> ranges = **new** ArrayList<List<Integer>>();

ranges.sort((x, y) -> Integer.*compare*(x.size(), y.size()));

**How to remove duplicate from a list,**

List<Integer> range = **new** ArrayList<Integer>(Arrays.asList(1, 1, 1, 3));

range = range.stream().distinct().collect(Collectors.*toList*());

**// range is {1, 3} now**

**How to provide data for the test of multi layer list (List<List<>>)?** //[1500, [3, 12, 14]]

List<List<Integer>> result = **new** ArrayList<List<Integer>>();

result.add(Arrays.*asList*(1500));

List<Integer> finalItems = Arrays.*asList*(3,12,14);

result.add(finalItems);

assertThat(obj.knapsackProblem(new int[][]{{1,2},{4,3},{5,6},{6,7}}, 10)).isEqualTo(result);

**How to generate empty List<List<>>?**

//[0, []]

List<List<Integer>> result = **new** ArrayList<List<Integer>>();

result.add(Arrays.*asList*(0));

result.add(Arrays.*asList*());

**How to copy one list to another without getting affected by reference change?**

List<Integer> itemIndexs = **new** ArrayList<Integer>();

itemIndexs.add(5);

List<Integer> highestIndexes = **new** ArrayList<Integer>();

highestIndexes = itemIndexs;

itemIndexs.clear(); // highestIndexes is also empty now because of reference

But when,

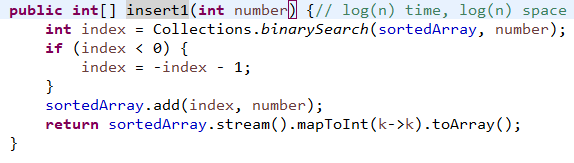
highestIndexes = **new** ArrayList<>(itemIndexs);

itemIndexs.clear(); // highestIndexes has the last values of itemIndexs

**When you need resizable queue then choose ArrayDeque. It is not thread safe and they are faster than stack and linkedlist,**

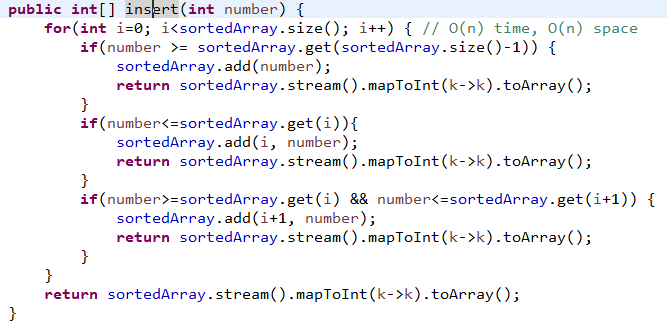
[ArrayDeque](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#ArrayDeque())<Integer> queue = new [ArrayDeque](https://docs.oracle.com/javase/7/docs/api/java/util/ArrayDeque.html#ArrayDeque())<Integer>();

[Fastest way to insert a value in sorted list in Java](https://stackoverflow.com/questions/29238427/fastest-way-to-add-a-value-in-the-middle-of-a-sorted-array-java). This is also called TimSort which is combination of Mergesort and Insertionsort.

****

**BinarySearch has O(logn) time and space complexity.**

**Other option is following which cost a bit more.**

****

**Top Sorting algorithm**

Merge sort is used in the Collections.sort() method Merge sort is a fast, stable sorting routine with guaranteed O(n\*log(n)) efficiency. When sorting arrays, merge sort requires additional scratch space proportional to the size of the input array.

When sorting *primitive values* by their natural order you won’t notice a difference as primitive values have no identity. Therefore, Quicksort is used for primitive arrays as it is slightly more efficient.

For objects you may notice, when objects which are deemed equal according to their equals implementation or the provided Comparator change their order. Therefore, Quicksort is not an option. So a variant of MergeSort is used, the current Java versions use ***TimSort***. This applies to both, Arrays.sort and Collections.sort

There are two most efficient sorting algorithms.  
1) Merge sort  
2) quick sort

So if you sort an array by merge sort or quick sort both required same time to sort that array.   
Both have O(n\*log n) complexity in best case and average case [NOTE:here log is to the base 2], in worst case quick sort have O(2^n) complexity and merge sort have O(Nlog N).  
So, in worst case merge sort gives better output.

But in practice, quick sort gives you better complexity, means it can sort an array in less time than merge sort.

**Now, for space (memory) complexity**

In merge sort we required one more array to merge and sort but in quick sort there is no need of another array. By space wise quick sort gives better result.

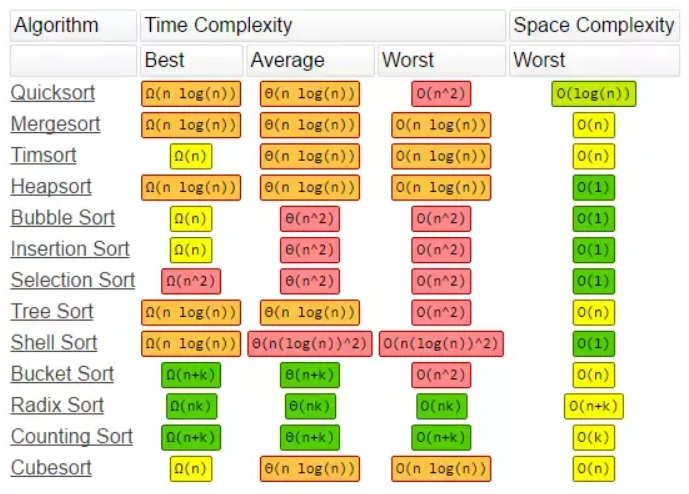
**Conclusion: *Quick sort***is best algorithm for sorting an array.

**For Unsorted array:**

Heap Sort (no extra space) = Merge Sort (O(n) extra space)>Quick Sort(O(n) space & O(n2)time in worst case)>Insertion Sort>Bubble Sort>Selection Sort

**For Sorted Array:**

Insertion Sort = Bubble Sort (O(n))>Heap Sort = Merge Sort>Quick Sort>Selection Sort.

****

**Java Graph Nodes or DFS handling or Multidimentional array**

How to store list of nodes consisting i and j,

List<Integer[]> unvisitedNeighbours = **new** ArrayList<Integer[]>();

How to keep track to visited nodes,

Boolean [][] visited = **new** Boolean[matrix.length][matrix[0].length];

Java has Stack to deal with DFS which allows push(), pop(),

Stack<Integer[]> nodesToExplore = **new** Stack<Integer[]>();

**For BFS, we can use queue,**

Queue<Node> queue = **new** LinkedList<Node>(); where we can do add() and poll()

**while**(!queue.isEmpty()) {

Node currentNode = queue.poll();

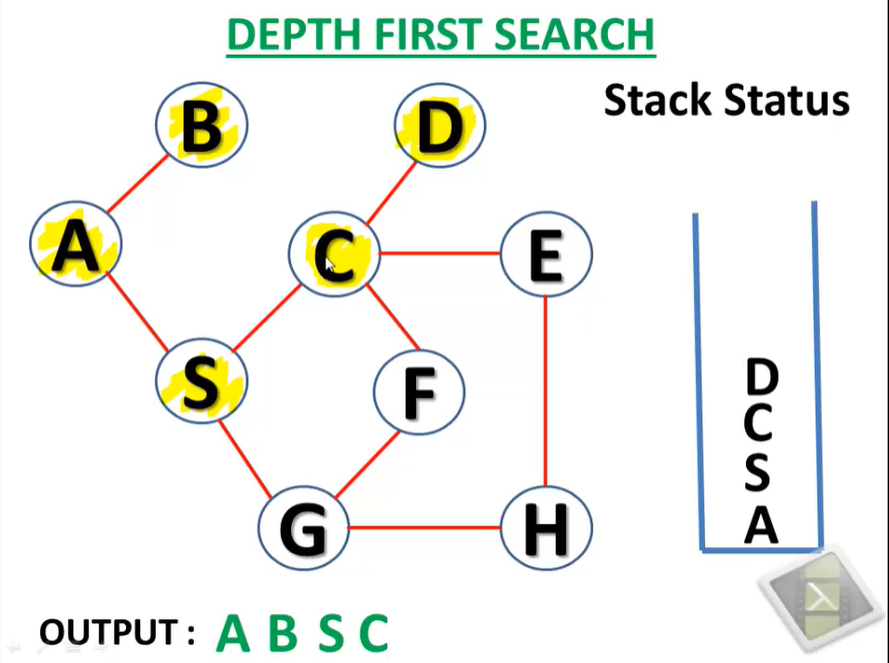
}

**DFS (Depth First Search) Stack based LIFO**

<https://www.youtube.com/watch?v=iaBEKo5sM7w>

It will push current node to the stack and goes alphabetically until leafs. It will pop it from the stack when it has no more connected node and the current node will came back to top node of the stack.

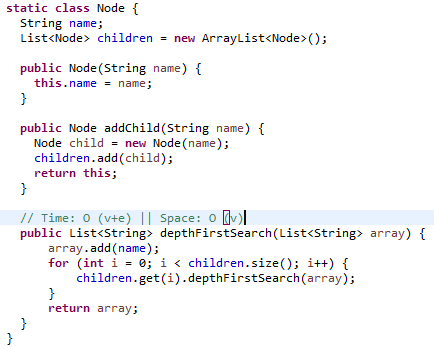
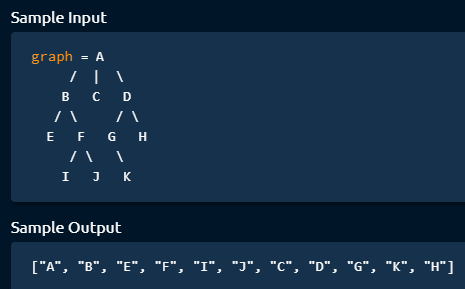
Starting with A, it will push A to the Stack and add in the output sequence. Alphabetically it will go to B as adjacent node and push B on the top of the Stack and add in the output sequence. B has no more adjacent node that is why it will come back to A and pop B from the stack. As a next option A will go to S and push S to the top of the Stack and add in the output sequence. From S, alphabetically C is the node to visit as next option and it will add C in the stack. C will direct to D as a next option and add D at the stack. As D has no more option to go, it will delete D from the stack.



**How to initialize and insert value in stack (DFS)? How to pick and process data from stack?** LIFO means books on top of each other. Here it is inserting adjacent value inside the loop.

**A solution of DFS for a graph is following: (Depth first search)**

To implement DFS for graph, we have to use recursive approach so that it can go until the depth of each node from the beginning. Notice that here in graph the nodes are connected with its child by List of node, not left right pointer as it was in tree.

** **

**How to generate a tree or graph by code?**

DFS.Node graph = **new** DFS.Node("A");

graph.addChild("B").addChild("C").addChild("D");

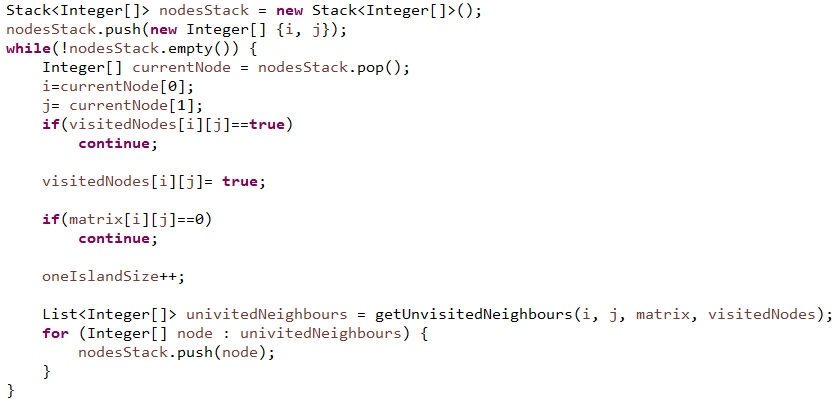
graph.children.get(0).addChild("E").addChild("F");

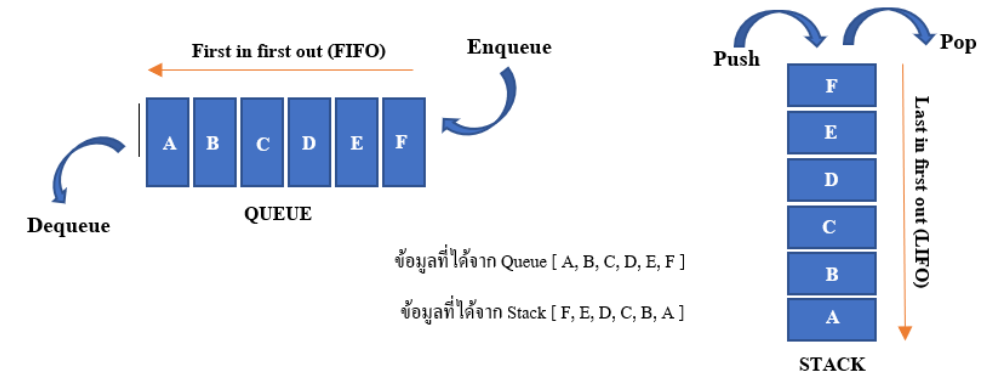
graph.children.get(2).addChild("G").addChild("H");

graph.children.get(0).children.get(1).addChild("I").addChild("J");

graph.children.get(2).children.get(0).addChild("K");

**This is a solution of DFS for adjacency matrix.**

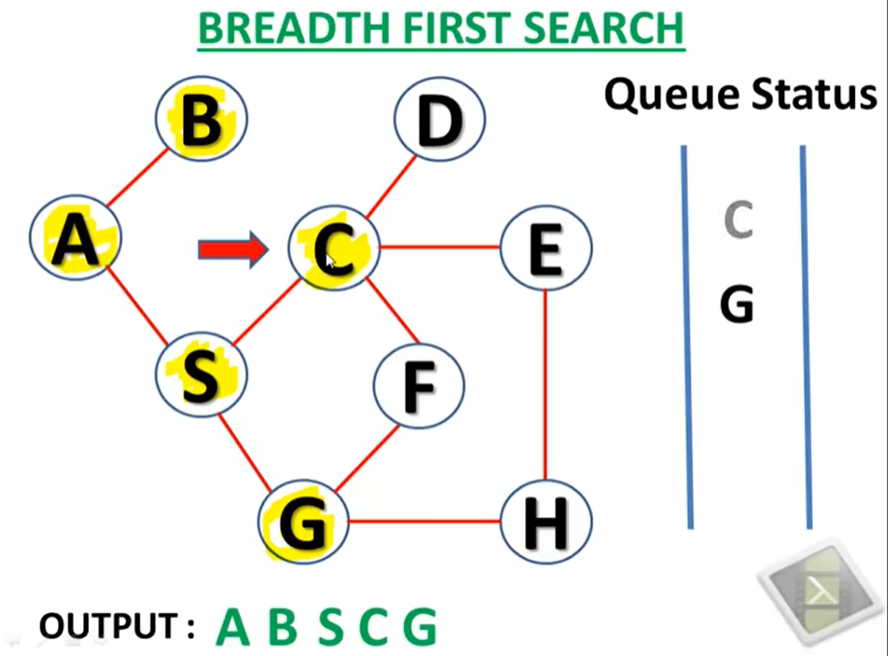




**BFS (Breath First Search) Queue based FIFO**

<https://www.youtube.com/watch?v=QRq6p9s8NVg>

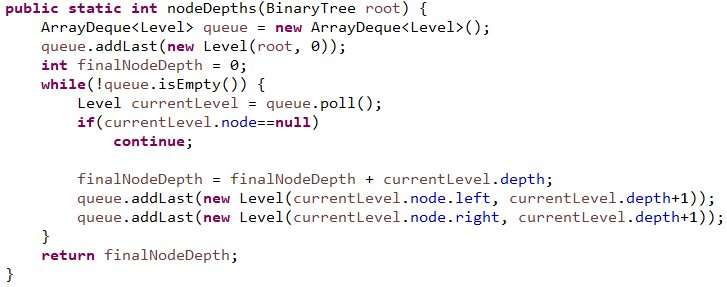
We have to mark output sequence first and mark it visited (Let start with A). So currently working node is A. As BFS we have to check all adjacent unvisited nodes. We have B and S here, so we will en-queue B and S alphabetically. And we mark it as visited and add it to the output sequence. Now we have to check first element of the queue, we have B here. So, we updated currently working node B and de-queue B. Now, we find that we have only A is an adjacent node which we have already visited. So, we will update the pointer to S and de-queue S. Now, we see S has adjacent node C and G. So, en-queue S and G and put them in the output sequence. So C is the current working node in the queue as it came first. C will be de-queued and as for adjacent from C, DEF node will be en-queued below G and will be added to the output sequence.



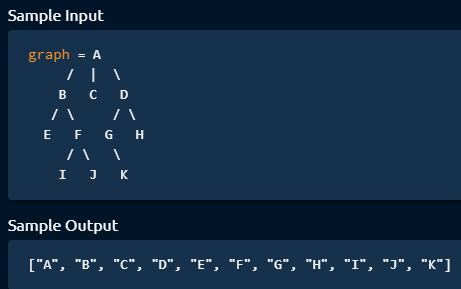
**Java has Queue to deal with BFS which allows addLast(), poll(),**

ArrayDeque<Level> queue = **new** ArrayDeque<Level>();

queue.addLast(**new** Level(root, 0));

****

**For graph, we can use BFS in following way,**

****

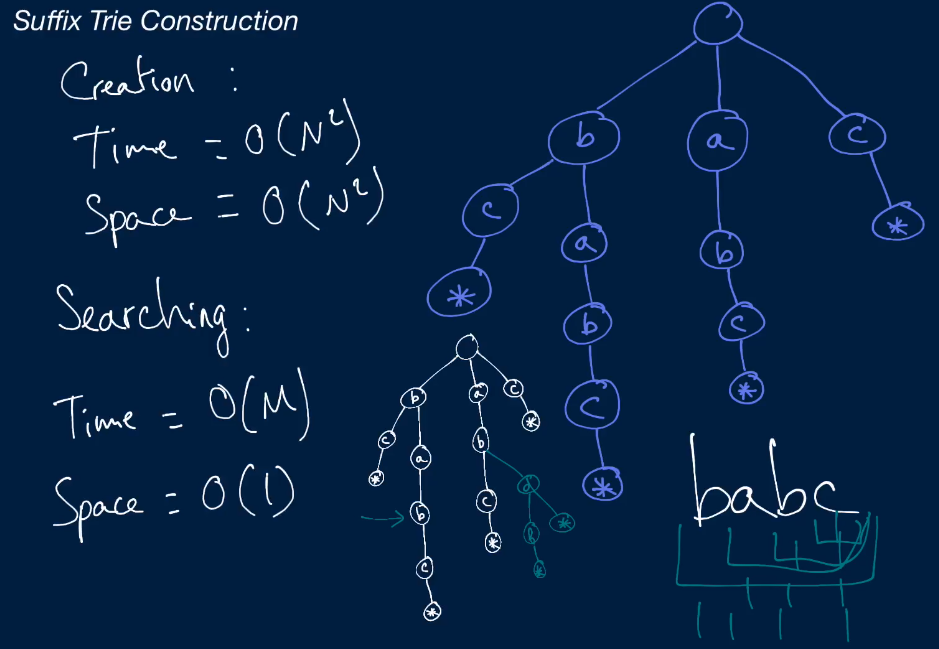
For BFS, we can use,

Queue<Node> queue = **new** LinkedList<Node>();

****

**Suffix Trie Construction**

Tries is basically used to construct string structure (with characters) which can be useful to make an extended type of search with comparatively less time complexity. First we need to generate Suffix trie with given String. Then we can do search on that.



The benefit we will get by constructing this trie is when we want to search the existence of a particular part of the string from the big string, it must search it from the beginning of the big string, but as every subset are stored on the trie, it will be able to find every element from the root if that string exists. So, it will reduce a big amount of search cost. But it requires suffix trie construction cost which O (n^2). Still it gives time complexity benefit over regular naïve techniques.

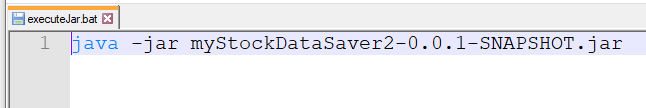
**Converting *Date* to *OffsetDateTime* is pretty simple. If our *Date* is in UTC, we can convert it with a single expression:**

|  |  |
| --- | --- |
|  | Date date = new Date();  OffsetDateTime offsetDateTime = date.toInstant()    .atOffset(ZoneOffset.UTC); |



**General Techniques**

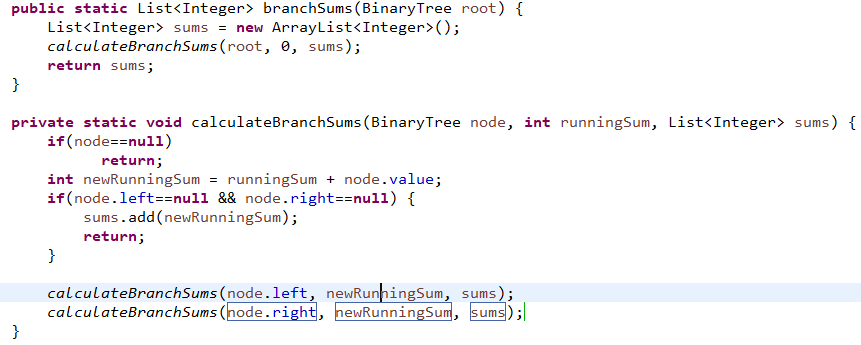
# [How to execute cmd command from text file?](https://stackoverflow.com/questions/19075543/how-to-execute-cmd-command-from-text-file)

****

Write the command and save the file with .bat, now a double click would already execute your command in cmd promt.

**Passing many variable by method parameter and get update without return, called pass object type by reference.**

In java, it is really good technique that you can pass any list or variable by method parameter which could be updated in that method and without return, you get the updated version of that method. That is object oriented programming and that’s way you can actually return more than one variables update. For example, the list **sums** was changed by second method many times but first method knows everything. The variable **runningSum** was initialized by the method call by 0 which is very interesting. When you need to do anything for tree leaf node, then this kind of recursive method call is very useful.



[**Can I pass a primitive type (Double, Integer, Boolean, etc) by reference in Java?**](https://stackoverflow.com/questions/3662288/can-i-pass-a-primitive-type-by-reference-in-java)

For Primitives are not passed by references (or objects for that matter) so no you cannot.

