**Java Trick and Techniques to make better**

**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.

**When any algorithm containing sorting, that means it will definitely need minimum O (nlogn).**

**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();

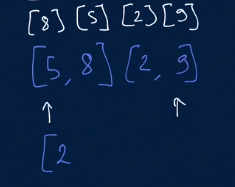
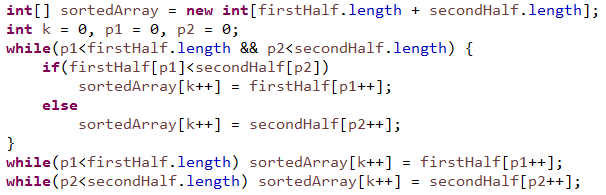
**How to separate parts of array,**

**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));

**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});

**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,**

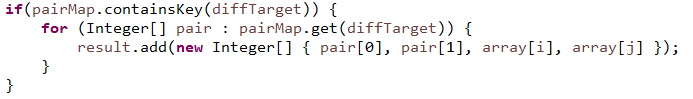
Integer[] newValuePair = **new** Integer[] {5, 7};

pairMap.put(12, newValuePair);

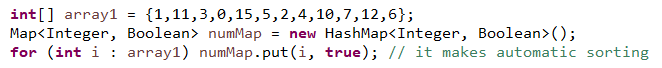
**How to add second Interger[] for that same key into that list,**

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

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

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

**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>();

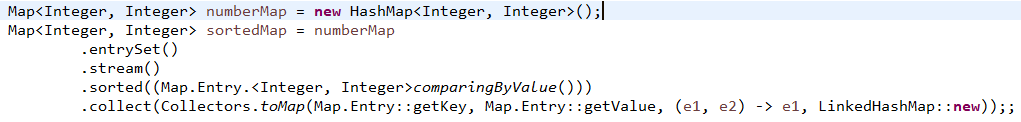
In single-threaded context, HashMap is the fastest option but it is not synchronized.

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<>();

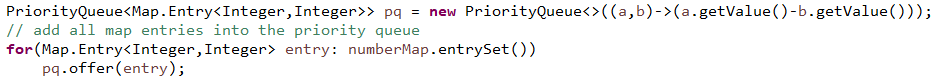
**When you need to sort a map by its value using lambda function using java 8,**



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,**

****

**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));

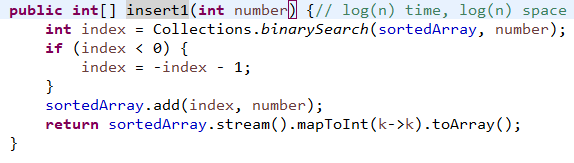
range = range.stream().distinct().collect(Collectors.*toList*());

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

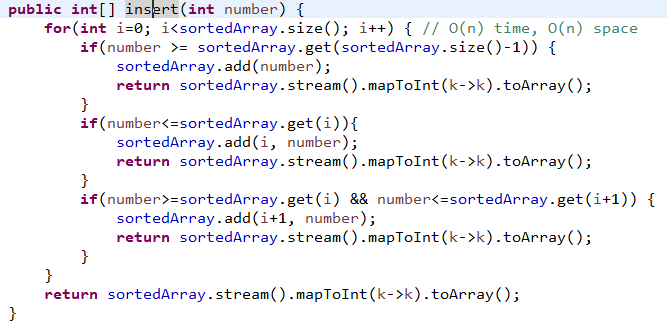
**When you need resizable queue then choose ArrayDeque. It is not thread safe and they are faster than stack and linkedlist,**

[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 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.

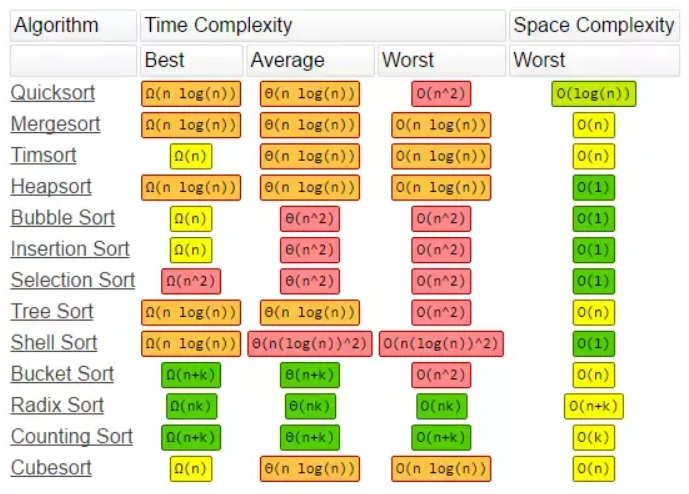
**Conclusion: *Quick sort***is best algorithm for sorting an array.

**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,

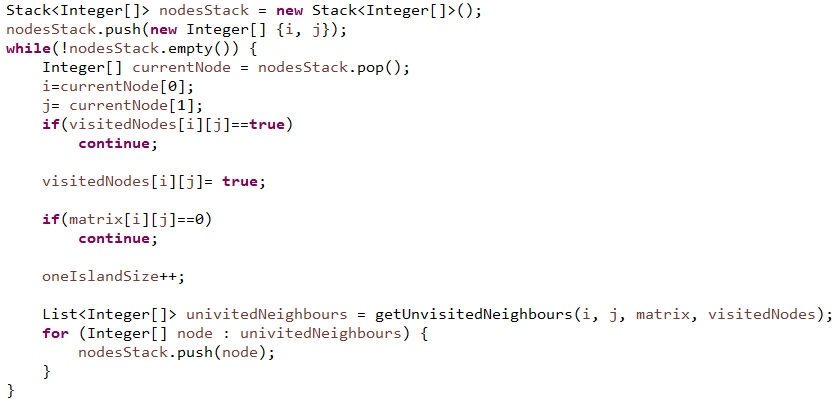
List<Integer[]> unvisitedNeighbours = **new** ArrayList<Integer[]>();

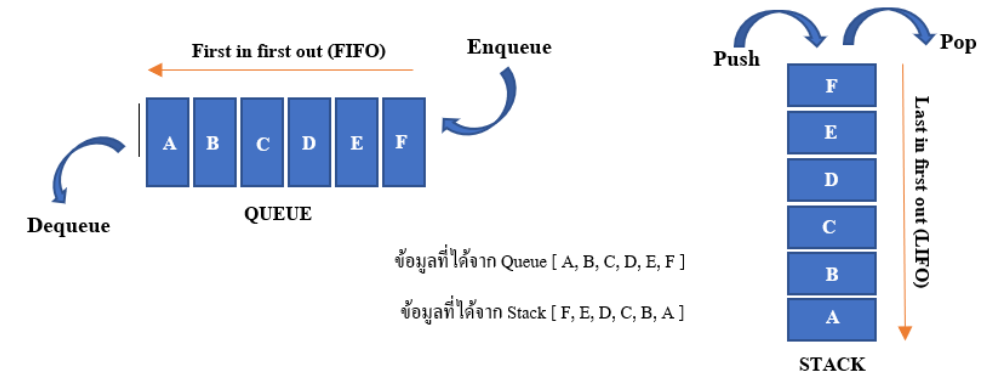
How to keep track to visited nodes,

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[]>();

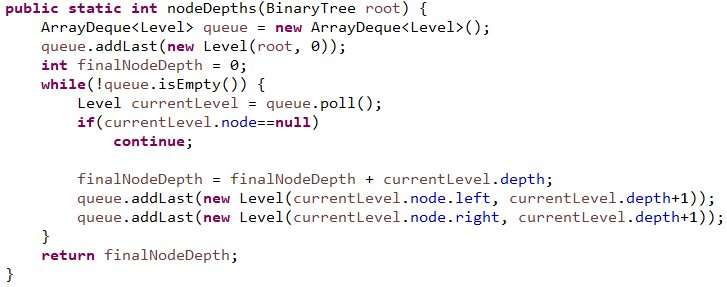
How to initialize and insert value in stack? How to pick and process data from stack? LIFO means books on top of each other. Here it is inserting adjacent value inside the loop. 



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

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

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

****

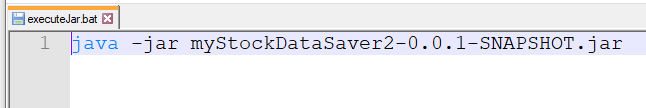
**Converting *Date* to *OffsetDateTime* is pretty simple. If our *Date* is in UTC, we can convert it with a single expression:**

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



**General Techniques**

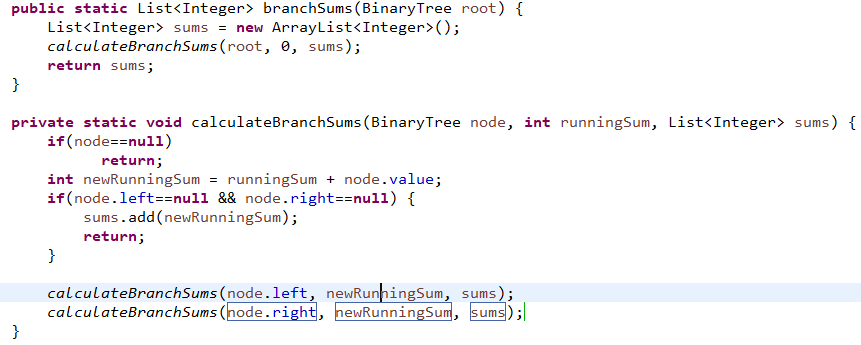
# [How to execute cmd command from text file?](https://stackoverflow.com/questions/19075543/how-to-execute-cmd-command-from-text-file)

****

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**

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.



**Is it only for list? I need be clear about it.**