Data Structures & Algorithms

[**Topics To Know**](#_bytwbwwzibaj) **2**

[**Test Cases**](#_xroyxl6sv42l) **3**

[**General Coding Notes:**](#_9jpw25kav019) **3**

[**String**](#_bahjky37fnew) **4**

[**Array**](#_fy4fkprpvcos) **7**

[**List & Collection**](#_u3kipncfm74k) **9**

[**Map/ Hash Table**](#_cd6ch9em1df) **10**

[**Misc Useful Code Blocks**](#_wlvgb9swx1vg) **12**

[**Sliding Window Pattern:**](#_li7boqntnbfe) **14**

[**Two Pointer Technique**](#_ie20huh97dxf) **16**

[Front Back Two Pointer](#_dc5yw03g2v8p) 16

[Fast & Slow Pointers](#_x156gyeyixq) 16

[**Merge Intervals**](#_3rk0zb6r8qfj) **17**

[**Classic Case**](#_9zv2c6uhd9na) **17**

[Overlap cases:](#_ynx9tb3f7jhg) 18

[**Stacks & Queue**](#_lu4t4r70jef0) **19**

[**Tree**](#_j2kjzomvy0tj) **21**

[**Breadth First Search**](#_ucxy9dgc0vlr) **22**

[**Depth First Search**](#_dwlo5k1cqptx) **23**

[**Recursion**](#_k79p2wz14xgq) **26**

[**Appendix A - Math**](#_swfdmylk0s2d) **27**

[Discrete Math Properties](#_terrvqsljryb) 28

[**Appendix - A: Resources**](#_ojpx86zfn4x6) **31**

[**Appendix - B: Great Leetcode Posts**](#_lw1ob6xw33sv) **31**

[**Appendix C: Patterns**](#_co6bqy3obaf4) **31**

[**Appendix D: Mindmap**](#_f0vvwlgfxb89) **33**

# 

# Topics To Know

1. Array & String
2. Dynamic Array/List
3. Map/Hash Table
4. Linked List
5. Stack
6. Queue
7. Trees
   1. Binary Tree
   2. Binary Search Tree
   3. Depth first search
   4. Breadth first search
   5. Tree Traversal
      1. Inorder traversal
      2. Pre order traversal
      3. Post order traversal
8. Trie
9. Priority queue
10. Graphs
    1. Edges
11. Recursion
12. Searching
    1. Binary search
13. Sorting
    1. Quick sort
    2. Merge sort
    3. Bubble sort
14. Backtracking
15. Greedy algorithms
16. Divide and conquer
17. Dynamic programming/Memoization
18. Greedy algorithms

# 

# Test Cases

* Input value = 0
* Input +ve number
* Input negative number
* Input empty
* Input null (note that primitives and arrays can’t be null)
* First item: Especially during iteration note that the first item is handled properly
* Last item: Especially during iteration note that the first item is handled properly
* Increment/decrement criteria: Note exactly how the variables, counters, etc change value.

# General Coding Notes:

* During iteration, generally the last step should be to pointer increment, otherwise, there’s risk of index out of bound error. E.g.

|  |
| --- |
| while (end < arr.length) {  //dom something  end++;  //following will cause index overflow  arr[end] = something;  } |

* Nested while loops are usually not necessary. For variable length resultset, try outer while loop with inner if/else.
* When the return type is an array, e.g. int[], but we’re not sure about the size of the return array, then use a list to hold the values (and adding items to that list). Finally, convert that list to an array and return.

|  |
| --- |
| //Array to list and list to array Integer[] arrTest1 ={1,2,3,4,5}; //Following will not work if arrTest1 was int[] List<Integer> listTest1 = Arrays.asList(arrTest1); Integer[] arrTest2 = listTest1.toArray(new Integer[0]); |

# String

|  |
| --- |
| /\*  Common String methods:  charAt(i) -> char in this String at i  contains(str) -> boolean  endsWith(str) -> boolean  equals(str) -> case sensitive equality  equalsIgnoreCase(str) -> case insensitive equality check  indexOf(str) -> first index where str starts (else -1)  lastIndexOf(str) -> last index where str starts (else -1)  length() -> number of chars  startsWith(str) -> boolean  substring(i, j) -> i (inclusive) to j (exclusive)  toLowerCase() -> new str with all lowercase  toUpperCase() -> new str with all uppercase  String.valueOf(number) -> number to string  Integer.valueOf(String) -> String to integer  Double.valueOf(String) -> String to double  String str = 1 + ""; -> Number + "" -> String.  \*/  public class StringCore {  @Test  public void testString() {  String string = " abcde ";  string.trim();  string.replace("b", "d");  string.toUpperCase();  string.toLowerCase();  string.contains("a");  string.charAt(0);  string.toCharArray();  }  public static void main(String[] args) {  String a = " New York "; //Initialized -> " New York "  a = a.trim(); //leading and trailing spaces removed -> "New York"  a = a.replace(' ', '\*'); // character replaced -> New\*York  a = a.replace("ew", "!!"); // seq of chars replaced -> N!!\*York  a = a.toUpperCase(); // -> N!!\*YORK  a = a.toLowerCase(); // -> n!!\*york  /\*  Note: String in Java is immutable, but StringBuilder is immutable  so, when a string is manipulated, a new string object is created in string pool and  the reference is changed to point to new object by the original object is unchanged.  We can prove that by comparing hashcode of the objects  TO compare value of two string objects, use equals()  \*/  //String can be viewed as a char[]  char[] charArray = {'L', 'o', 'c', 'k'};  String c = new String(charArray); // -> Lock  //Reverse String c in place  char[] d = c.toCharArray();  int size = d.length;  for (int i = 0; i < size / 2; i++) {  char temp = d[i];  d[i] = d[size - i - 1];  d[size - i - 1] = temp;  }  //-> kcoL  //Reversing with two pointers  char[] charArray2 = {'T', 'e', 'n', 't'};  String e = new String(charArray2);  int m = 0;  int n = charArray2.length - 1;  while (m < n) {  char temp = charArray2[m];  charArray2[m] = charArray2[n];  charArray2[n] = temp;  m++;  n--;  }  // -> tneT  //Other string methods  String z = "Sample String...";  boolean bool = z.contains("sa"); // bool -> false  bool = z.startsWith("Sa"); // bool -> true  bool = z.startsWith("ple", 3); //offset = index of the first letter. bool -> true  bool = z.endsWith("g..."); // bool -> true  bool = z.equalsIgnoreCase("SAMPLE String..."); // -> true  String zSubString = z.substring(1, z.length() - 1); // inclusive start and exclusive start -> ample String..  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  //VARIOUS TRICKS//  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  //Is a given character letter or digit  String str = "This is 2019!";  bool = Character.*isLetterOrDigit*(str.charAt(0)); // true  int num = 5;  String strNum = "" + num; //converts to string  String someNum = "33";  int someNumInt = Integer.*parseInt*(someNum); // -> 33  System.*out*.println(someNum);  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  // STRING BUILDER //  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//  //For multiple manipulations, use StringBuilder  StringBuilder b = new StringBuilder("California");  b.append(", CA"); // -> California, CA  b.reverse(); // -> AC ,ainrofilaC  b.reverse(); // -> California, CA  b.deleteCharAt(b.length() - 1); // -> California, C  b.delete(b.indexOf(","), b.lastIndexOf("C") + 1); // same as -> b.delete(10, 12+1); -> California  }  } |
|  |

# Array

|  |
| --- |
| @Test public void arrayTest() {  int[] arr = {1,2,3,4,5} ;//=> Initialization of a an int array  //arr[x] = y ; Assigns y to the xth index of arr   int [] arr2 = new int [arr.length]; //array of length 5  Arrays.fill(arr2, 0);// fills array2 with 0 => [0,0,0,0,0]   //Array copying  // All methods use System.arraycopy() under the hood.  //Creates shallow copy for non-primitive arrays  Arrays.copyOf(arr, arr.length); //From index 0 till length. => [1,2,3,4,5]  Arrays.copyOfRange(arr, 0, arr.length);  // Arrays.copyOfRange(src, startInclusive, endExclusive)  arr.clone(); // Using the object.clone() method. => [1,2,3,4,5]  int [] arr6 = new int[arr.length];  System.arraycopy(arr, 0, arr6, 0, arr.length);  // System.arraycopy(src, srcStart, dst, dstStart, length) => void   //Shallow copy example  String[] strA = {"Alan", "Becky", "Chloe"};  String[] strB = strA;  strA[0] = "newAlan";// strB[0] is now also newAlan   //Sorting.  // Java uses Merge sort or Time sort depending on the version  // O(n log n) time and O(n) space  int[] arr7 = {7,5,8,3,8,9};  Arrays.sort(arr7); //[3, 5, 7, 8, 8, 9] void method.   int [][] arr2dA = {{5,1}, {7,1}, {8,1}, {2,1}};  Arrays.sort(arr2dA, (a,b) -> Integer.compare(a[0], b[0]));  //Arrays.sort(arr2dA, Comparator.comparingInt(a -> a[0]));  //Collection.sort(theList, (a,b) -> Double.compare(a.get(0), b.get(0)));  List<List<Double>> list = Arrays.asList(  Arrays.asList(5.1, 2.1, 6.7),  Arrays.asList(4.1, 2.1, 6.7),  Arrays.asList(6.1, 2.3, 6.7));  Collections.sort(list, (a,b) -> Double.compare(a.get(0), b.get(0)));  //[[4.1, 2.1, 6.7], [5.1, 2.1, 6.7], [6.1, 2.3, 6.7]] }  public static void main(String[] args) {  /\*  int[] arr = {1,2,3,4,5} => Initialization of a an int array  arr[x] => returns item at xth index  arr[x] = y => Assigns y to the xth index of arr  Arrays.copyOf(srcArr, length) => new array  Arrays.copyOfRange(srcArr, startIndex, endIndex) => new array  System.arraycopy(src, srcStart, dst, dstStart, length) => void  Arrays.sort(arrSrc) => void  Arrays.fill(src, val) => void  \*/    //Create the DS  String[] strArr = {"Asia", "Europe", "Africa", "America", "Australia"};   //First and last items  String firstItem = strArr[0]; //Asia  String lastItem = strArr[strArr.length - 1]; //Australia  System.out.println(String.format("First iItem: %s \nLast item: %s", firstItem, lastItem));   //Copy array (from inclusive, to exclusive)  String[] copyArr = Arrays.copyOfRange(strArr, 0, strArr.length);  System.out.println("Array copy: " + Arrays.toString(copyArr) + "\n"); //"Asia", "Europe", "Africa", "America", "Australia"   //String as char array. Reverse string  String str = "FooBar";  char[] charArr = str.toCharArray();  for (int i = 0; i < charArr.length / 2; i++) {  char temp = charArr[i];  charArr[i] = charArr[charArr.length - i - 1];  charArr[charArr.length - i - 1] = temp;  }  String reversedString = new String(charArr);  System.out.println("Reserved String: " + reversedString + "\n"); //raBooF   //Arrays.copyOf(srcArr, length) => returns new array. if length > srcArray, then the rest will be filled with 0  //Arrays.copyOfRange(srcArr, startIndex, endIndex) => returns new array. Start inclusive, end exclusive. If end> srcArr.length, rest filled with 0  //System.arraycopy(src, srcStart, dst, dstStart, length) => void method. Copies items from src to dst.  //Arrays.sort(arrSrc) => Void. Sorts the array. n \* log n  //Arrays.fill(src, val) => Void. Fills array with the given val.  int[] arr = {1, 2, 3, 4, 5};   //Copy all but the first and last items to another array  int[] arr2 = Arrays.copyOfRange(arr, 1, arr.length - 1);  System.out.println(Arrays.toString(arr2));   //Fill an array of size 5 with 0 and then with 1  int[] arr3 = new int[5];  Arrays.fill(arr3, 0);  System.out.println(Arrays.toString(arr3));  Arrays.fill(arr3, 1);  System.out.println(Arrays.toString(arr3)); // 1,1,1,1,1   //Copy the first three items from arr to the last three positions of arr3  System.arraycopy(arr, 0, arr3, arr.length - 3, 3);  System.out.println(Arrays.toString(arr3)); // 1,1,1,2,3   //Copy the first three items of arr into a new array  int[] arr4 = Arrays.copyOf(arr, 3);  System.out.println(Arrays.toString(arr4));// 1,2,3   //Sort an array  int[] arr5 = {3, 4, 2, 6, 9, 2, 5, 3};  Arrays.sort(arr5);  System.out.println("arr5 (sorted): " + Arrays.toString(arr5));   //Compare elements in 2d arrays  int[][] arrayA = new int[][]{{4,2,3}, {9,7,8}};  System.out.println(Arrays.toString(arrayA));  //two-dimensional array in Java is nothing but an array of a one-dimensional array,  //you can also create a two-dimensional array where individual one-dimensional arrays have different length, } |
|  |

# 

# List & Collection

# 

# Map/ Hash Table

|  |
| --- |
| @Test  public void mapTest() {  Map<String, String> map = new HashMap();  map.put("k1", "v1");  map.put("k2", "v2"); //Keys can't be duplicate or null. Values can be  map.get("k1"); // => v1. value of key k1  map.getOrDefault("k3", "defaultVal"); // => defaultVal, since k3 absent  map.putIfAbsent("k3", "v3"); //=> v3. Added k3,v3 (if k3 were absent)  map.remove("k2"); // => v2. Removed the key value pair  Map.Entry<String, String> entry = new AbstractMap.SimpleEntry<>("k5", "v5");  map.entrySet(); // [k1=v1, k3=v3]. Set of the entries.  map.keySet(); // [k1, k3]. Set of keys  map.values();// [v1, v3]. Collection of values. Will include all values, including duplicates.  map.containsKey("k2"); //true  map.containsValue("v4");//false  map.isEmpty(); //false  map.size(); //  //map.putAll(map2); // void. add all key-values of map2 to map  map.clear();// void. Removes all key-values of map.  }  public static void main(String[] args) {  //Create map and add items to map  Map<String, Object> myMap = new HashMap<>();  myMap.put("Continent", "North America");  myMap.put("Population", 579000000);  myMap.put("AreaMilesSq", 95400000);  myMap.put("IncomePerCapitaUSD", 49000);  myMap.put("RedundantInfo", "someRedundantInfo");  //Put a key-val, only if this key doesn't exist  myMap.putIfAbsent("IncomePerCapitaUSD", 50000);  //Remove a key-value pair  myMap.remove("RedundantInfo");  //Method 1 - Iterate over the map and get the keys and values  List<String> listMapKeys = new ArrayList<>();  List<Object> listMapValues = new ArrayList<>();  for (Map.Entry<String, Object> e : myMap.entrySet()) {  listMapKeys.add(e.getKey());  listMapValues.add(e.getValue());  }  //Method 2- Iterate over the map and get the keys and values  // map.keySet() -> returns set  // map.values() -> returns collection  Set<String> listMapKeys2 = myMap.keySet();  List<Object> listMapValues2 = new ArrayList<>(myMap.values());  //Method 3 - Iterate over the map and get the keys and values  //Collection.Iterator()  Iterator<Map.Entry<String, Object>> mapIterator = myMap.entrySet().iterator();  List<String> listIteratorKeys = new ArrayList<>();  List<Object> listIteratorValues = new ArrayList<>();  while (mapIterator.hasNext()) {  //note:iterator.next() progresses the iterator to next item. So, following is buggy  //listIteratorKeys.add(mapIterator.next().getKey());  //listIteratorValues.add(mapIterator.next().getValue().toString());  Map.Entry<String, Object> e = mapIterator.next();  listIteratorKeys.add(e.getKey());  listIteratorValues.add(e.getValue().toString());  }  //Create a countMap using getOrDefault(key, defaultValue)  char[] charArray = {'a', 'e', 'o', 'a', 'e', 'z'};  Map<Character, Integer> countMap = new HashMap<>();  for (int i = 0; i < charArray.length; i++) {  countMap.put(charArray[i], countMap.getOrDefault(charArray[i], 0) + 1);  }  // If the following lists contain extra items (compared to myMap), remove them  List<String> listKeysToCompare = new ArrayList<>(Arrays.*asList*("Continent", "Population", "Tada", "AreaMilesSq"));  List<Object> listValuesToCompare = new ArrayList<>(Arrays.*asList*("North America", 95400000, 49000, 121212));  for (int i = 0; i < listKeysToCompare.size(); i++) {  if (!myMap.containsKey(listKeysToCompare.get(i))) listKeysToCompare.remove(i);  }  for (int i = 0; i < listValuesToCompare.size(); i++) {  if (!myMap.containsValue(listValuesToCompare.get(i))) listValuesToCompare.remove(i);  }  //Are all items in a list distinct - method 1  List<String> sampleList = Arrays.*asList*("Apple", "Banana", "Orange", "Peach", "Banana");  Map<String, Integer> sampleCountMap = new HashMap<>();  for (int i = 0; i < sampleList.size(); i++) {  sampleCountMap.put(sampleList.get(i), sampleCountMap.getOrDefault(sampleList.get(i), 0) + 1);  }  boolean isDuplicateList1 = sampleCountMap.size() == sampleList.size();  //Are all items in a list distinct - method 2  Set<String> uniqueListItems = new HashSet<>(sampleList);  boolean isDuplicateList2 = uniqueListItems.size() == sampleList.size();  //For equality, order does not matter for set. set1.equals(set2) = true  //List can't be compared with Set, incontrovertible type.  //But the list can be cast to a set and compared  Set<String> set1 = new HashSet<>(Arrays.*asList*("Dhaka", "NYC", "Tokyo", "Budapest"));  Set<String> set2 = new HashSet<>(Arrays.*asList*("NYC", "Dhaka", "Tokyo", "Budapest"));  boolean setContains = set1.contains("NYC");//true  set1.add("NYC"); //-> This will leave the set unchanged as NYC is already in set  set1.remove("New"); //-> Won't change anything as the set doesn't contain New  set1.equals(set2); // -> true  List<String> list3 = Arrays.*asList*("NYC", "Dhaka", "Tokyo", "Budapest");  new HashSet<>(list3).equals(set1);// -> true  //To use Set vs. Dictionary  //Note that Set is good with distinct items, but  //if the items need to be removed and a count is required, use dictionary  } |

# Misc Useful Code Blocks

|  |
| --- |
| //Iterate over a collection Iterator<Integer> it = queue.iterator(); while (it.hasNext()) {  System.out.println("Queue item: " + it.next()); }  //Iterate over a map for (Map.Entry<String, String> e : map.entrySet()) {  System.out.format("%s : %s\n", e.getKey(), e.getValue()); }  //Adding items in a reverse order LinkedList<String> list = new LinkedList<>(); list.add(0,"FirstItem"); list.addFirst("NewFirstItem"); //[NewFirstItem, FirstItem]  //Declare multiple variables in the same line  int a = 10, b = 20;  //If else execution differences  public static void testIfElse() { //executes all the statements that are satisfied by condition  for (int i = 0; i < 10; i += 2) { //0, 2, 4, 6, 8  if (i < 5) System.*out*.printf(" %s < 5, \t", i);  if (i < 6) System.*out*.printf(" %s < 6, \t", i);  if (i < 7) System.*out*.printf(" %s < 7, \t", i);  if (i < 8) System.*out*.printf(" %s < 8, \t", i);  System.*out*.println(" Final statement");  }  }  //Result of testIfElse()  /\*  0 < 5, 0 < 6, 0 < 7, 0 < 8, Final statement  2 < 5, 2 < 6, 2 < 7, 2 < 8, Final statement  4 < 5, 4 < 6, 4 < 7, 4 < 8, Final statement  6 < 7, 6 < 8, Final statement  Final statement  \*/  public static void testIfElse2() {//Executes only the first statement, from top to bottom, that is satisfied  for (int i = 0; i < 10; i += 2) { //0, 2, 4, 6, 8  if (i < 5) System.*out*.printf(" %s < 5, \t", i);  else if (i < 6) System.*out*.printf(" %s < 6, \t", i);  else if (i < 7) System.*out*.printf(" %s < 7, \t", i);  else if (i < 8) System.*out*.printf(" %s < 8, \t", i);  else System.*out*.println(" Final statement");  }  }  //Order of execution in a conditional statement (if/while). Left to right  //while (start < end && arr[start] == arr[start + 1])  //NOT the same as while (arr[start] == arr[start + 1] && start < end )  //Esp with indices (start+1) be careful to check when it gets out of bounds |

# Sliding Window Pattern:

**Clues:**

* Usually array, string (or a LinkedList) problems
* Find or calculate something among all the
  + Contiguous items
  + **subarray**
  + **substring**
* Contagious sublists of a given size
* The subarray is the “window”
* Window can be **fixed** or **variable (while loop)** depending on the problem statement.

**Coding notes:**

* Fixed window (if statement) vs variable window (while statement)
* Size of window = end - start +1
* For statement to iterate across the array/string etc.
  + In each iteration, **end pointe**r increases by one
  + While/if statement to check if the start pointer needs to move and the window needs to be shrunk
* For max/min use **Math.max(maxCount, end - start + 1)**; -> be very careful where this statement needs to be put (Do NOT put it while the success condition is false).
* Use a **Map for counting frequency**, (e.g. number of unique elements in the window).
* Note that **Window size == end - start + 1 == (sum of map.getvalues())**
* The keyset, map.ketSet(), represents the unique set of values.

**Standard template (fixed length)**

|  |
| --- |
| //Find the average of all contiguous subarrays of size 'K' in it. public static double[] findAverages(int K, int[] arr) {  double sum = 0; //variable to be tracked for processing  int start = 0; // window start  int end = 0; //window end  double[] sumArray = new double[arr.length - K + 1];   for (end = 0; end < arr.length; end++) { //grow from right, end++  sum += arr[end];  if (end >= K - 1) { // shrinking condition  sumArray[start] = sum / K;  sum -= arr[start];  start++; //shrink from left-> last step in while loop  }  }  return sumArray; } |

**Standard template (variable length)**

|  |
| --- |
| //Find the length of the longest substring in it with no more than K distinct characters. public static int findLength(String str, int k) {  //Pseudo code  //Sliding window of variable length  //slide end pointer and keep count of the frequency of distinct letters  //if map.size() > k, keep sliding start pointer to reduce window size  //keep sliding start pointer until map size <= K  //Keep track of the max window size (result)  //Finally add input validation   if (str == null || str.length() == 0 || str.length() < k) {  throw new IllegalArgumentException("Input is invalid");  }   Map<Character, Integer> charMap = new HashMap<>();  int maxCount = 0;  int start = 0;  int end = 0;  for (end = 0; end < str.length(); end++) {  //In each iteration, slide the end pointer by one (always happens)  charMap.put(str.charAt(end), charMap.getOrDefault(str.charAt(end), 0) + 1);  //Continue to shrink the window as long as successful criteria not met  while (charMap.size() > k) {  if (charMap.get(str.charAt(start)) == 1)  charMap.remove(str.charAt(start));  else charMap.put(str.charAt(start), charMap.get(str.charAt(start)) - 1);  start++; //last step in while loop...move start pointer  }  maxCount = Math.max(maxCount, end - start + 1); //needs to put outside while since inside while, success condition == false  }  return maxCount; } |

# Two Pointer Technique

## Front Back Two Pointer

* Sorted array, linked list
* Classic use: items x and y such that f(x,y) == target value
* If not sorted, consider sorting them first, e.g. Arrays.sort(arr) => n long n
* These problems are not necessarily contiguous (so not sliding window), e.g. two numbers that add upto X.
* Can have multiple arrays as input, e.g. merge two sorted arrays.
  + Consider merging from the ends as an option esp if one array has extra elements to accommodate at the end.
* General steps:
  + Sort if not sorted (n \* log n additional time complexity)
  + Left pointer, left = arr[0]
  + Right pointer, right = arr[arr.length -1]
  + f(right, left) = result
  + Based on result, figure out which pointer needs to move
  + Keep moving them until result == desired result or if
  + Left > right (crossing condition) -> means no result found

## Fast & Slow Pointers

* Fast pointer moves ahead of the slow pointer.
* Classic use:
  + Skip duplicates.
  + Requires the array to be sorted first (if unsorted) -> n log n
  + First item is always unique, e.g. for (i == 0 || arr[i] != arr[i-1])
* Movement pattern:
  + Fast pointer (j) iterates through the list/array/string
  + Slow pointer (i) moves ONLY when success criteria is met
  + E.g. Duplicate removal in sorted array. Fast pointer iterates through the entire list, while slow pointer moves only when the unique element is encountered.
* Check: While returning the value of slow pointer, what what the value of i would be,e.g. Is it some index or length (i+1) ?

# Merge Intervals

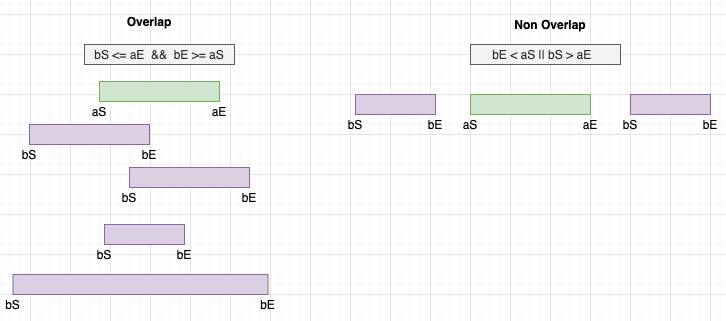
# 

## Classic Case

* Sort the arrays by the first item ( n log n)
* Declare a list of results, List<int[]>
* Var newInterval tracks resultLists's last item
* Iterate over the sorted arrays (int[] interval:intervals)
* Compare interval[0] vs. newInterval[1]
* interval[0] > newInterval[1] => non overlapping
* If non-overlapping, add interval to result
* If overlapping, merge newInterval to interval
* Merge -> set newInterval[1] to Math.max(interval[1], newInterval[1])
* Convert resultList to array, results.toArray(new int[0][])

|  |
| --- |
| //Given a collection of intervals, merge all overlapping intervals. public int[][] merge(int[][] intervals) {   List<int[]> results = new ArrayList<>();  if (intervals.length < 2) {  return intervals;  }   Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));  //newInterval refers to the last item in results list  int[] newInterval = intervals[0];  results.add(newInterval);  for (int[] interval : intervals) {  //non overlap case  if (interval[0] > newInterval[1]) {  newInterval = interval;  results.add(newInterval);  } else {  //overlap - change just the end value of newInterval  newInterval[1] = Math.max(newInterval[1], interval[1]);  }  }  //list to array conversion  return results.toArray(new int[0][]); } |

## Overlap cases



b will overlap with a:

If (bStart <= aEnd && bEnd >= aStart)

b will not overlap with a:

If (aStart > bEnd || aEnd < bStart)

Note: Intersection is the opposite of merge. Sample code below

|  |
| --- |
| List<int[]> result = new ArrayList<>(); for (int[] a : A) {  for (int[] b : B) {  if (a[1] < b[0]) {  //No chance of further overlap so break out of inner for loop  break;  }  //a overlaps with b (a left, a right, a subset)  else if ((a[0] >= b[0] && a[0] <= b[1]) || (a[1] >= b[0] && a[1] <= b[1])) {  int[] intersection = new int[]{Math.max(a[0], b[0]), Math.min(a[1], b[1])};  result.add(intersection);  }  //b subset of a  //Some of the following checks are redundant  //else if ((b[0] >= a[0] && b[0] <= a[1]) || (b[1] > a[0] && b[1] <=a[1])){  else if (b[0] > a[0]) {  int[] intersection = new int[]{Math.max(a[0], b[0]), Math.min(a[1], b[1])};  result.add(intersection);  }  } } |

# 

# Stacks & Queue

Stack:

* LIFO
* Depth first search

Queue:

* FIFO
* Breadth first search

|  |
| --- |
| @Test  public void testStackQueue() {  //Queue  Deque<String> stack = new ArrayDeque<>();  stack.push("sv1"); //void  stack.push("sv2");  stack.push("sv3");  stack.peek(); //sv3. Return the last item  stack.pop(); //sv3. Also removes v2 from stack.  stack.size(); // 2, since sv3 popped  stack.isEmpty(); // false  Arrays.*toString*(stack.toArray()); //[sv2, sv1]  stack.clear(); //removes all items  stack.isEmpty(); //true  //Stack  Deque<String> queue = new ArrayDeque<>();  queue.offer("qv1"); //true, returns boolean  queue.offer("qv2");  queue.offer("qv3");  queue.peek(); // qv1. Doesn't remove anything  queue.poll(); // qv1. Removes qv1 from the queue  queue.size(); // 2  queue.isEmpty();//false  queue.clear(); //removes all items  queue.isEmpty();// true  //Other available methods, not to be used for stack/queue  Deque<String> deque = new ArrayDeque<>();  deque.addFirst("d1"); //void  deque.addLast("d2"); //void  deque.getFirst(); //d1  deque.getLast(); //d2  deque.removeFirst(); //d1  deque.removeLast(); //d2  deque.contains("d2"); //false  } |
|  |

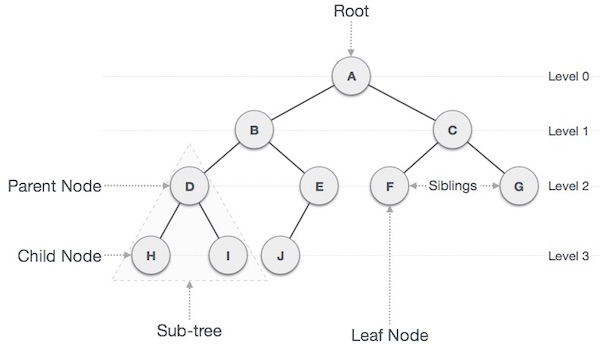
Java recommends using the Deque interface for stacks and queues.

**ArrayDeque:** is a special kind of a growable array that allows us to add or remove an element from both sides. An ArrayDeque implementation can be used as a Stack (Last-In-First-Out) or a Queue(First-In-First-Out).

# 

# Tree

* Hierarchical nonlinear data structure
* Nodes connected by edges
* No cycles
* Node:
  + Internal Node: Node with at least one child
  + Root Node: Top node/prime ancestor of a tree
  + Leaf node: Node(s) with no children
* Depth (of a node): Number of edges between a node and the root.Root node has depth = 0
* Level (of a node): Number of edges+1 between a node and the root. Level = Depth + 1
* Height (of a tree): Maximum depth of any node in that tree.
* Breadth (of a tree): The number of leaves of a tree.
* Binary Tree: Tree where each node has at most two children. Left child and right child.
* Binary Search Tree:
  + Binary tree where leftChildNode < Node < rightChildNode
  + Each of the nodes form BST themselves.
* Tree traversal: Process of visiting (processing) the nodes of a tree exactly one.
  + Breadth first search:
    - Finding the shortest path between two nodes.
    - Lever order traversal- visit all of the nodes of a given depth before moving on to the next depth.
  + Depth first search:Starting with root, visit all of the nodes of a given branch as deep as possible before backtracking.
    - In order traversal: LeftChildNode - Node - RightChildNode => Will give nodes in a sorted manner
    - Pre order traversal: Node - LeftChildNode - RightChildNode
    - Post order traversal: LeftChildNode - RightChildNote - Node



# Breadth First Search

* **Algorithm for traversing** or searching tree or graph data structures.
* It explores **all the nodes at the present depth** before moving on to the nodes at the next depth level.
* Can be implemented using a queue (FIFO)

**BFS Template:**

|  |
| --- |
| //Class to conduct BFS - print the items of each level as a list public class BreadthFirstSearch {  //Create a queue to hold the nodes  //Check if the root is null. If not null, add the root  //Do the following until queue is empty (while !queue.isEmpty())  //Get queue size. Iterate over that queueSize -> represents the level size  //Poll from queue and point root to that. Add root to levelList  //Add the left and right children of root to the queue, if they exist  public static List<List<Integer>> levelOrderTraversal(TreeNode root) {  List<List<Integer>> result = new ArrayList<>();  if (root == null) return result;  Queue<TreeNode> queue = new LinkedList<>();  queue.offer(root);   while (!queue.isEmpty()) {  List<Integer> currentLevel = new ArrayList<>();  int queueSize = queue.size();  for (int i = 0; i < queueSize; i++) {  root = queue.poll();  currentLevel.add(root.val);  if (root.left != null) queue.offer(root.left);  if (root.right != null) queue.offer(root.right);  }  result.add(currentLevel);  }  return result;  } |

# 

# Depth First Search

* Algorithm for traversing or searching tree or graph data structures
* Uses the idea of backtracking.
* It explores all the nodes by going forward if possible or uses backtracking.
* Can be implemented using a stack (LIFO)

|  |
| --- |
| public class DepthFirstSearch {   //Var needs to be outside method for recursions.  //otherwise would get reinitialized each time.  public static List<Integer> result = new ArrayList<>();   public static List<Integer> inOrderTraversalRecursive(TreeNode currentNode) {  //Implicitly the call stack is used here for LIFO.  //left - node - right  if (currentNode != null) {  inOrderTraversalRecursive(currentNode.left);  result.add(currentNode.val);  inOrderTraversalRecursive(currentNode.right);  }  return result;  }   public static List<Integer> preOrderTraversalRecursive(TreeNode currentNode) {  if (currentNode != null) {  //node - left - right  result.add(currentNode.val);  preOrderTraversalRecursive(currentNode.left);  preOrderTraversalRecursive(currentNode.right);  }  return result;  }   public static List<Integer> postOrderTraversalRecursive(TreeNode currentNode) {  if (currentNode != null) {  //left - right - node  postOrderTraversalRecursive(currentNode.left);  postOrderTraversalRecursive(currentNode.right);  result.add(currentNode.val);  }  return result;  }   //--------ITERATIVE APPROACH WITH EXPLICIT STACK--------//   public static List<Integer> preOrderTraversalIterative(TreeNode root) {  List<Integer> result = new ArrayList<>();  if (root == null) return result;  Stack<TreeNode> stack = new Stack<>();  stack.push(root);   while (!stack.isEmpty()) {  root = stack.pop();  result.add(root.val);  if (root.right != null) stack.push(root.right);  if (root.left != null) stack.push(root.left);  }  return result;  }   public static List<Integer> postOrderTraversalIterative(TreeNode root) {  LinkedList<Integer> result = new LinkedList<>();  if (root == null) return result;  Stack<TreeNode> stack = new Stack<>();  stack.push(root);   while (!stack.isEmpty()) {  root = stack.pop();  result.addFirst(root.val); //adding in a reverse order  if (root.left != null) stack.push(root.left);  if (root.right != null) stack.push(root.right);  }  return result;  }   public static List<Integer> inOrderTraversalIterative(TreeNode root) {  List<Integer> result = new ArrayList<>();  Stack<TreeNode> stack = new Stack<>();  if (root == null) return result;   while (!stack.isEmpty() || root != null) {  while (root != null) {  stack.push(root);  root = root.left;  }  root = stack.pop();  result.add(root.val);  root = root.right;  }  return result;  } } |
|  |

# 

**Example: Depth first search to do some work using recursion.**

|  |
| --- |
| public class EC\_PathWithGivenSequence {   public static boolean findPath(TreeNode root, int[] sequence) {  //DFS  //at each level compare the node val with seq val  //if not equal then false  //if at the leaf and the values are equal, then true   if (root == null) return (sequence.length == 0);  int level = 0;  return helper(root, sequence, level);  }   private static boolean helper(TreeNode root, int[] sequence, int level) {  if (root == null) return false;  //if (root.val != sequence[level]) return false; //This is not entirely right...what if the sequence is too small?  if (level >= sequence.length || root.val != sequence[level]) return false;  if (root.right == null && root.left == null && sequence[level] == root.val && sequence.length == level + 1)  return true;  level++;  //This part is critical. OR condition since any one success is good enough.  return (helper(root.right, sequence, level) || helper(root.left, sequence, level));  }  } |

# Recursion

* Technique of problem solving by **breaking the problem** into smaller instances of itself until a **base case.** Then we **combine the results** of the subproblems to get an answer to the original problem.
* A recursive problem is defined in terms of:
  + **Bases case:** One or more cases where given input, output can be **calculated directly.**
  + **Recursive step**: result is computed using one or more recursive calls to the function itself but r**educing the complexity/size of the inputs** with each call, and getting closer to the base case.
* Base case usually refers to some empty item: t**he empty string, the empty list, the empty set, the empty tree, zero, etc.**
* All recursive problems can be solved iteratively (using loops), but some problems are naturally recursive:
  + Recursive data: E.g. file systems, binary trees etc.
  + Recursive problems: When the problem can be defined by itself.
* **Helper methods:** 
  + Useful private methods that help with the implementation of the recursive problem.
  + Useful when the calculation requires some variable to keep track of something at each recursive step.
  + Usually requires a different set of parameters than the main function.
  + Helper func takes care of the base case and recursive steps.
* **Shell/Main function:** 
  + Defines the input items (what do we need).
  + Calls the helper function with the defined attributes.
  + Return the output.
* **Disadvantage:** 
  + May take up more memory in the call stack.
  + Call stack is limited in size
  + That places a limit on the size of the recursive problem that can be solved.

|  |
| --- |
| public class Palindrome {  //Steps- declare the vars that we need.  //Call the helper func with those things. return the result.  public static boolean isPalindromeWithHelper(String s) {  s = s.toUpperCase();  char firstLetter = s.charAt(0);  char lastLetter = s.charAt(s.length() - 1);  return isPalindromeHelper(firstLetter, lastLetter, s);  }   //private helper func. Recursive  //Especially useful when a var/list etc. need to be maintained  private static boolean isPalindromeHelper(char firstLetter, char lastLetter, String s) {  if (s.length() == 0 || s.length() == 1) return true;  if (firstLetter != lastLetter) return false;  String reducedString = s.substring(1, s.length() - 1);  return isPalindromeHelper(reducedString.charAt(0), reducedString.charAt(reducedString.length() - 1), reducedString);  }   //Direct Recursion  public static boolean isPalindromeDirect(String s) {  s = s.toUpperCase();  if (s.length() <= 1) return true;  if (s.charAt(0) != s.charAt(s.length() - 1)) return false;  return isPalindromeDirect(s.substring(1, s.length() - 1));  } } |

# 

# Appendix A - Math

## Discrete Math Properties

* **Reflexive :** for any reference value x, **x.equals(x)** should return true.
* **Symmetric :** for any reference values x and y, if x.equals(y) should return true then y.equals(x) must return true.

**IF x=y , THEN y=x**

* **Transitive :** for any reference values a, b, and c, if a.equals(b) returns true and b.equals(c) returns true, then a.equals(c) should return true.

**if x=y and y=z , then x=z**

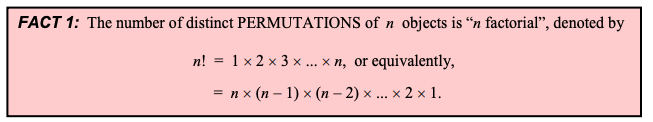
* **Consistent :** for any reference values a and b, multiple invocations of a.equals(b) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.
* **Substitution:** If x=y , then x may be replaced by y in any equation or expression.

Combinatorics

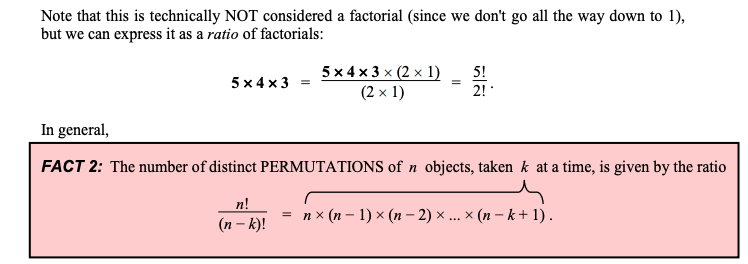
**Permutation**: Order matters. X, Y is not the same as Y, X.

**E.g. 5 people.**  How many different ways of getting a portrait (exact order matters here).

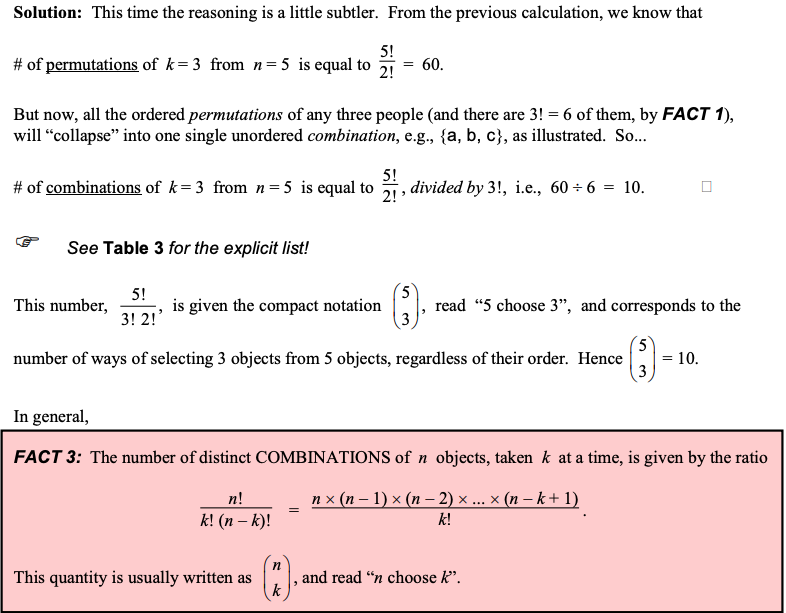
5 X 4 X 3 X2 X 1 => 5! => 120

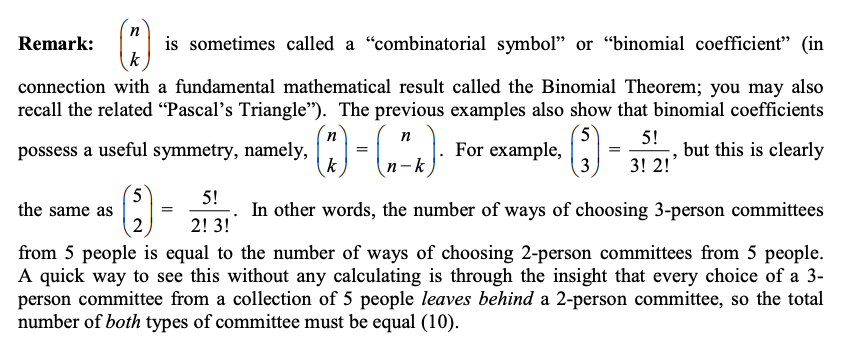


**E.g. 5 people, 3 slots.** How many different ways of getting a portrait (exact order matters here).



**E.g. 5 people, group of 3.** How many possible groupings? (order doesn’t matter)





# Appendix - A: Resources

* Collections cheat sheet: <https://courses.cs.washington.edu/courses/cse143/17su/exams/final/cheat_sheet.pdf>
* Data structure interviewcake: <https://www.interviewcake.com/data-structures-reference>
* Tree intro and traversal: <https://towardsdatascience.com/4-types-of-tree-traversal-algorithms-d56328450846>

# Appendix - B: Great Leetcode Posts

* <https://leetcode.com/discuss/interview-experience/581596/2020-xp-facebook-google-amazon-microsoft-dropbox-paypal-hackerrank>
* <https://leetcode.com/discuss/general-discussion/458695/dynamic-programming-patterns>
* <https://leetcode.com/discuss/interview-experience/424540/Google-or-L5-or-MTV-or-Oct-2019-Offer>
* <https://leetcode.com/discuss/interview-experience/418395/Amazon-or-SDE1-or-Seattle-or-Oct-2019-Offer>
* <https://leetcode.com/discuss/general-discussion/390646/I-Leetcoded-(And-So-Can-You!)>
* <https://leetcode.com/discuss/career/216554/From-0-to-clearing-UberAppleAmazonLinkedInGoogle>
* <https://leetcode.com/discuss/career/229177/My-System-Design-Template>

# Appendix C: Patterns

1. Sliding window
   1. Fixed window
   2. Variable window
2. Two pointers
   1. Pointers from two ends, esp for sorted arrays
   2. Fast and slow pointers, e.g. Cycles in a linked list.
3. Merge intervals
4. Cyclic sort
5. Breadth first search
   1. Using queue
   2. Level order search
6. Depth first search
   1. Recursively
   2. Iteratively using stack
7. Modified Binary Search Trees
8. Two Heaps
   1. Max heap
   2. Min heap
9. Subsets
10. Bit Manipulation
    1. Bitwise XOR
11. Top K elements
12. K way merge
13. Dynamic programming
14. Topological sort
15. Greedy algorithms
16. Dive and Conquer
17. Sorting
    1. Quick sort
    2. Merge sort
    3. Insertion sort
18. Trie
19. Union Find
20. Backtracking
21. Sorting algorithms:
    1. Merge sort
    2. Quick sort
    3. Bubble sort
    4. Insertion sort
    5. Radix sort

# Appendix D: Mindmap

If input array is sorted then

- Binary search

- Two pointers (two discrete items from the array/string)

- Sliding window (contiguous arrays, substring, subarray)

If asked for all permutations/subsets then

- Backtracking

If given a tree then

- DFS

- BFS

If given a graph then

- DFS

- BFS

If given a linked list then

- Two pointers

If recursion is banned then

- Stack

If asked for maximum/minumum subarray/subset/options then

- Dynamic programming

If asked for top/least K items then

- Heap

If asked for common strings then

- Map

- Trie

Else

- Map/Set for O(1) time & O(n) space

- Sort input for O(nlogn) time and O(1) space