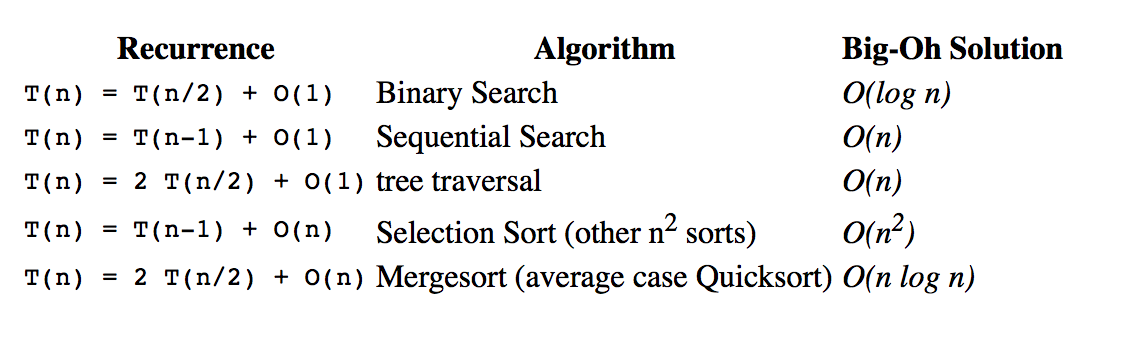
**V. Algorithm**

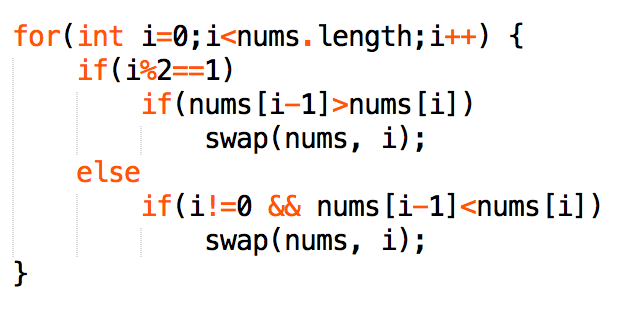
1. **Time Complexity**

* **Master Methd**
* Works only: T(n) = aT(n/b) + Θ(n^c), where a >= 1 and b > 1
* 3 cases:
  + If c < Logba, then T(n) = Θ(n^(Logba))
  + If c = Logba, then T(n) = Θ(n^bLog n)
  + If c > Logba, then T(n) = Θ(f(n))
* **Recursive Tree Method**
* draw a recurrence tree and calculate the time taken by every level of tree
* sum the work done at all levels
* **Common recursion complexity**



1. **Array:**

* **Arrays.binarySearch(arr, num):**
* find a number’s insert index of a sort array; return index if exist, o/w return (-(insertion point) - 1).
* When write our own binary search, something we should use left = mid instead of left = mid – 1, remember to change while loop to while (left < right-1) {}, o/w cause infinite loop
* **Kth largest problem solution:**
* Arrays.sort(): O(N log N) time + O(1) memory
* PriorityQueue: O(N log K) time + O(K) memory
* Selection algorithm: O(N) best case / O(N^2) worst case time + O(1) memory
* but after randomize input array, O(N) time guarantee (See Appendix 1.)
* **Wiggle sort problem**
* nums[0] <= nums[1] >= nums[2]…
* just remember to keep each pair valid from left to right



* **Range Addition**
* Brute force take O(N^2), but O(N) is achievable.
* Just store each value at start index, and -value at end index + 1
* for example [1 , 3 , 2] , [2, 3, 3] (length = 5)

res[ 0, 2, 0, 0 -2 ] 🡪 res[ 0 ,2, 3, 0, -5] 🡪 sum = [0, 2, 5, 5, 0]

* **Product of Array Except Self (Without Division)**
* For example, given [1,2,3,4], return [24,12,8,6].
* Not division allowed, brute force need O(n^2), but O(n) time is achievable
* By scan the array twice. First, calculate the running product of the part before the current number. Second, do the same from end
* **Array/String Pattern Match**
* When only care COMBINATION (**Moving window Problem** Leetcode-438**)**
* use 1 map to record pattern.
* Initiate ‘count = number part’.
* move window, if exit element matches a part, count++; if incoming element matches a part, count--.
* whenever occurs ‘count == 0’, there is a pattern match.
* When also care SEQUENCE (Word Pattern)
* Str="dog cat cat dog" matches pattern "abba"
* Use 2 map, iterate through all elements, for example, one map “dog -> 0”, another map “a -> 0”
* Any difference in map1.get() & map2.get() lead to match failure.
* **Rotate matrix**
* Just transpose + mirror
* **Remove duplicate from sorted array**
* Duplicate is allowed at most k times. So simply compare n with nums[i - k]

**for** (**int** n: nums)

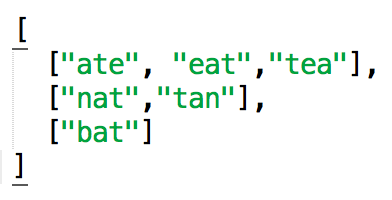
**if** (i < k || n > nums[i - k])

         nums[i++] = n;

* **Subarray Sum problem**
* When want to know SUM[i, j], just calculate SUM[0, i - 1] & SUM[0, j]
* With help of HashMap, we can optimize from O(n^2) to O(n)
* **Get Subset/Permutation List**
* Subset I (without duplicate)
* Barely backtracking with O(N!) time; just remember to create new object before add to result: list.add(new ArrayList<Integer>(temp));
* Subset II (contains duplicate)
  + Step 1, Arrays.sort(array);
  + Step 2, skip duplicate use: while (i > start && nums[i -1] == nums[i]) i++; (若前面的duplicate用过了，跳过)
* Combination Sum II (contains duplicate)
  + Each number can be used only once
  + The same skip method as Subset II
* Permutation I (no duplicate)
  + like [1,2,3][1,3,2][2,1,3]…
  + use: if (tempLst.contains(nums[i])) continue; with O(N!) time
  + or if(used[i]) continue; which save O(n) time, but require extra O(n) space
  + 只有permutation需用到boolean[] used, 因为每次选择number都不得不从所有number里，而不是剩下的里面选
* Permutation II (contains duplicate)
  + Arrays.sort() （只要contains duplicate都需要先sort）
  + Just as I, but if (used[i] || i <0 && nums[i] == nums[i-1] && ! used[i-1]) (若前面的duplicate没用过，跳过)

1. **Hash Table:**

* **Group Strings Problem**
* Use HashMap to store <word, index> of course, but if we can find a ‘uniform’ <pattern, index>.
* For example, call Arrays.sort(str.toCharArray()), make ‘aet’ the pattern for [‘ate’, ‘eat’, ‘tea’] as index = 0

****

* **Intersection of 2 arrays**
* Easy with Hash, but there is follow up.
* **Follow up 1:** If only nums2 cannot fit in memory. Put all elements of nums1 into a HashMap, read chunks of array that fit into the memory, and record the intersections.
* **Follow up 2:** If both nums1 and nums2 are so huge that neither fit into the memory, sort them individually (external sort), then read 2 elements from each array at a time in memory, record intersections.
* **Follow up 3:** Know how to use: map.getOrDefault(key, V); map.putIfAbsent(key, V)
* **Follow up 4:** If known limit range of element, allows O(1) space (etc. alphabet, use char[26] rather than Hash map)
* **Count Prime**
* checks if a single number is prime:

for (i = 2; i\*i <= num; i++) {

if (num % i == 0) return false

}

* count number of prime in range of (0 - n), O(n^2) time

for (i = 2 🡪 n)

if (notPrime[i] == false)

count++;

for (int j = 2 -> i\*j < n)

notPrime[I\*j] = true

* **Top K Frequency Element**
* Whenever wish to sort by frequency,
* **step 1** is to create a map<element, frequency>
* **step 2** is to do bucket sort

List<Integer>[] bucket = new List[nums.length + 1];

* **step 3** is to iterate list from tail, and create final result list

if(bucket[pos] != null)

res.addAll(bucket[pos]);

1. **Linked List:**

* **Sort linked list**
* Use merge sort, and use fast-slow pointer to find mid
* **Sorted List to BST**
* recursion on 2 child node with base case: if(head == tail) return null;
* Use faster-slow pointers to find the mid node of a list
* node = new TreeNode(slow.val)

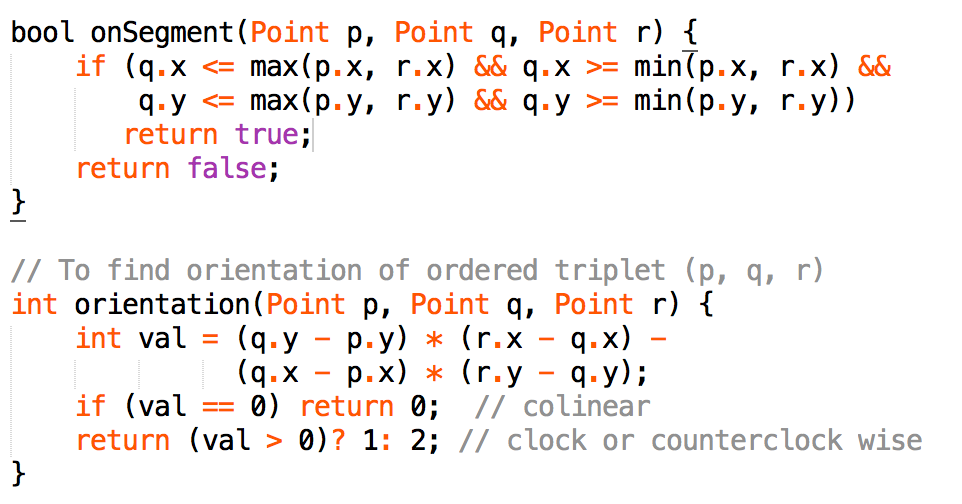
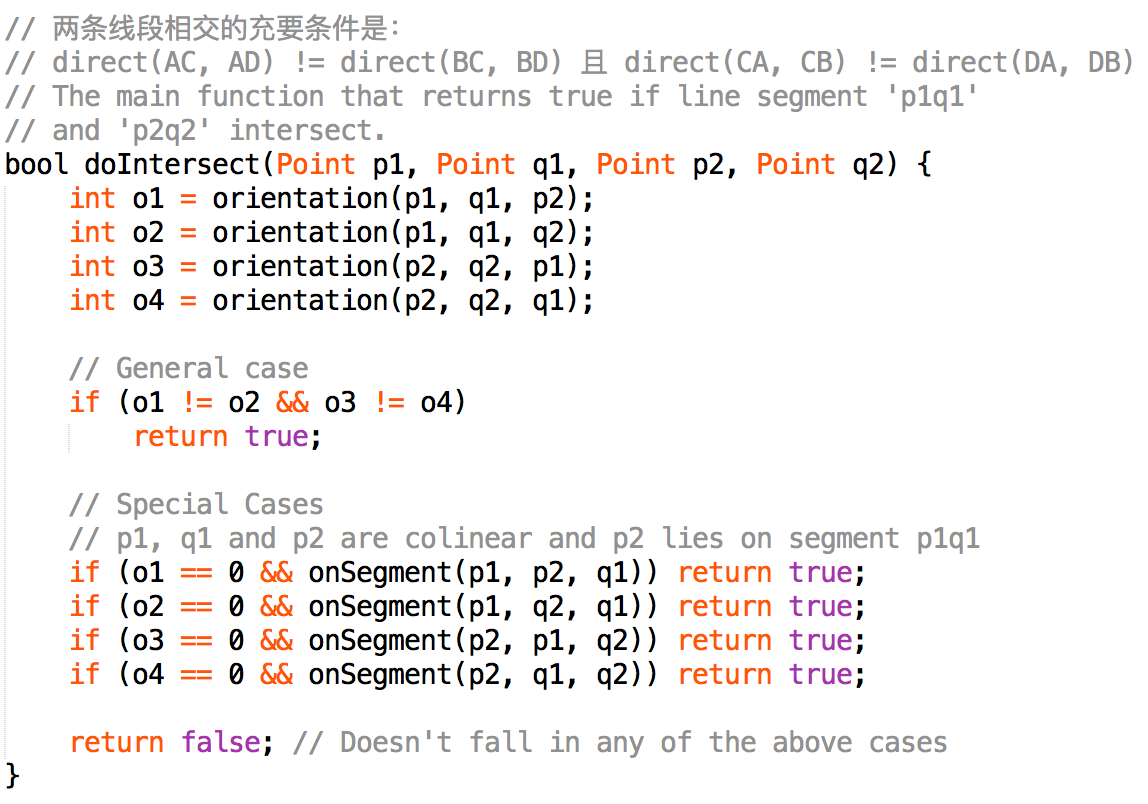
node.left = toBST(head, slow);

node.right = toBST(slow.next, tail);

* **Partition List/ Separate Odd and Even Node**
* Simply use 2 list to store each ‘type’, and finally concat
* Always need a Dummy Node to solve the List problem
* **Merge K problem**
* Merge K sorted List use Priority queue
* Merge 2 list from head, just iterative calculate
* However, when from tail like this: (7->2->4) + (5->6) = 7->8->0, consider use stack
* **Special case**
* Pascal's Triangle can be calculate in O(k) space by add(1) on left side, and then modify original list with A[j] += A[j + 1]
* For bug free, always check if node != null; be careful at List<Node> or List<Integer>

1. **Math:**

* **Rotate/update array**
* Always easy to get O(n^2) brute force solution, however can be optimized
* The key lay on limited change, for example, ‘all increment except one’ -> only one decrement
* Or like rotation, if decreased by sum, only one element changed
* **Check 2 line across**

****

* **Ugly Number Problem (**HARD**)**
* Ugly numbers are positive numbers whose prime factors only include given prime list int[] primes , and we want to know the Nth ugly number.
* Solution:
* use similar merge method as merge sort, to get every ugly number from the 3 subsequences
* Every step we choose the smallest one, and move one step after, including nums with same value.
* for example primes = {2,3,5}, image there are 3 sorted list as below

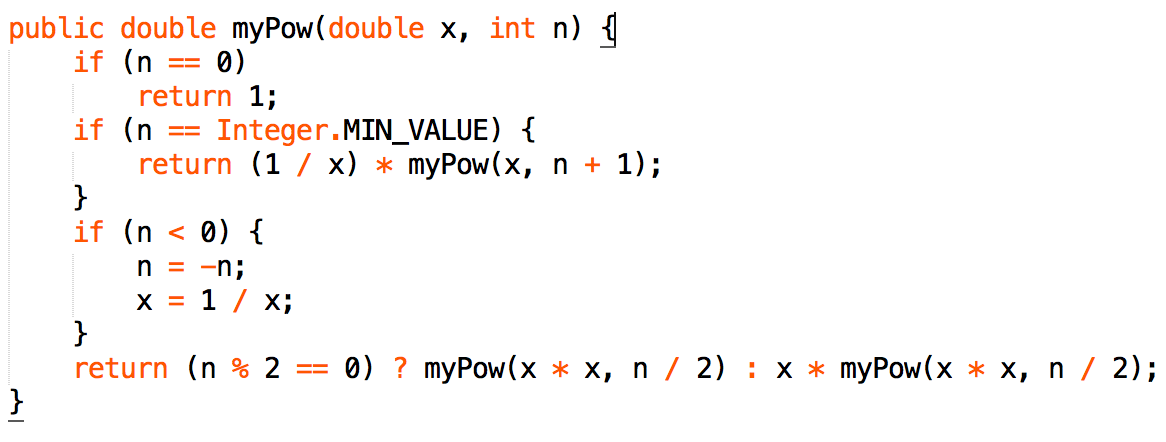
♣ (1) 1×2, 2×2, 3×2, 4×2, 5×2, …

♣ (2) 1×3, 2×3, 3×3, 4×3, 5×3, …

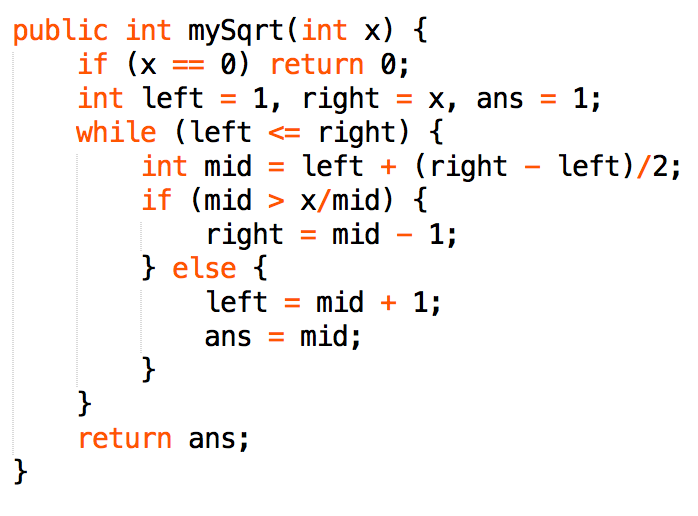
♣ (3) 1×5, 2×5, 3×5, 4×5, 5×5, …



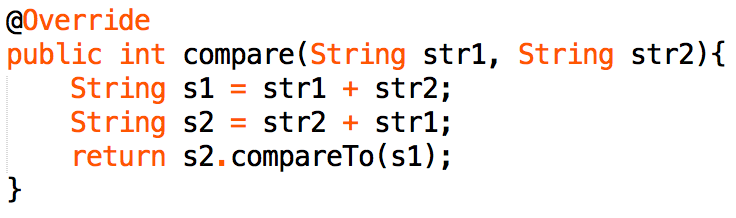
* **Implement Pow()**
* Consider base case n = 0
* Consider n < 0
* Consider n is even/odd
* Consider data overflow n = Integer.MIN\_VALUE

****

* **Implement Sqrt()**
* Just modify Binary search

****

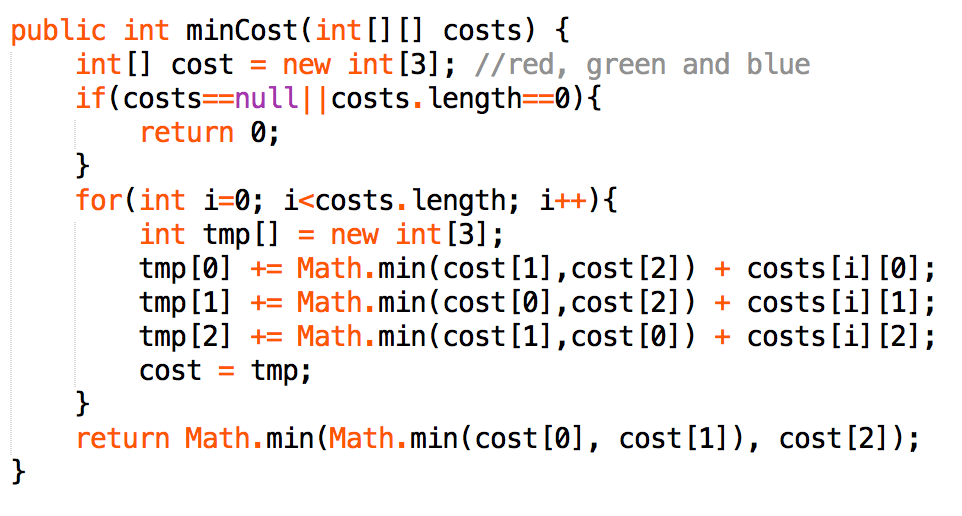
* **Largest Number Combination**
* **[23, 9], we can have maximum 923**
* **basically sort the array with new comparator in a smart way**

****

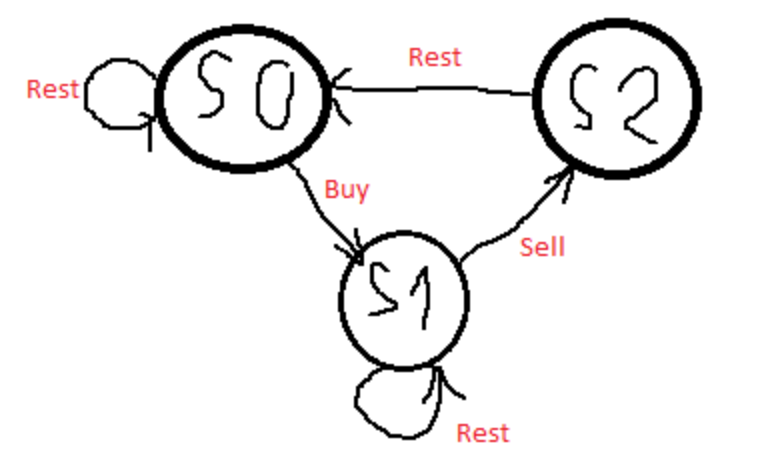
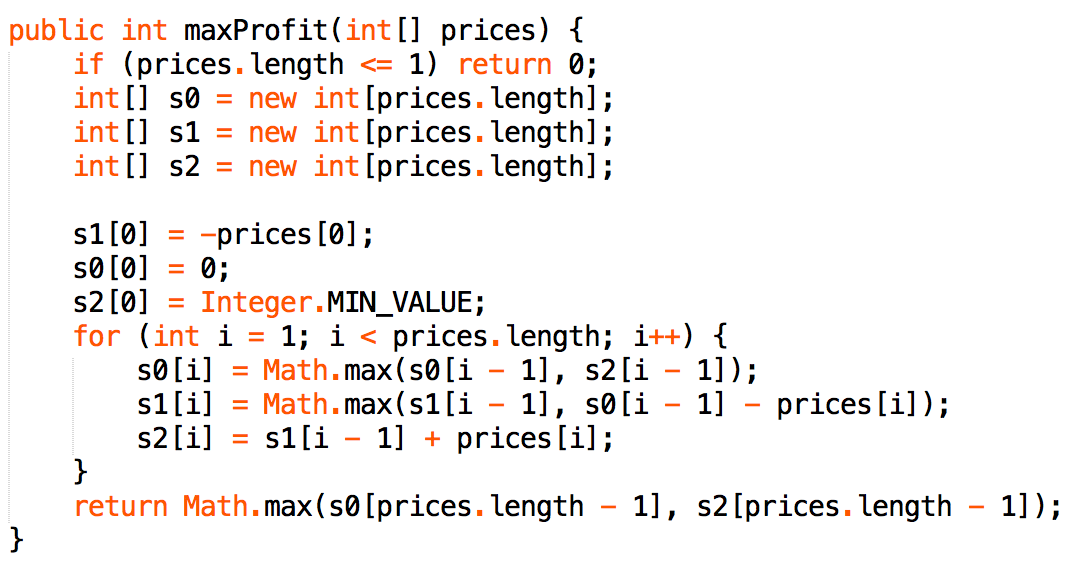
* **Best Meeting Point**
* Whether it is 1 dimension or 2 dimension, divide it into 1 dimension
* The key is to find the median of the array because it guarantee shortest path
* Find median could use both sorting(O(nlogn)) or quick select (O(n))

1. **Dynamic Programing**

* **DP problem definition**
* Optimal substructure
* Overlap sub-problem
* **Paint House**
* cost[n = 0->2] the cumulative cost when house N paints color[n]



* house painting similar problem (no more than 2 adjacent same color), calculate same color and different color condition separately
  + 1. int temp = diffColorCounts;
    2. diffColorCounts = (diffColorCounts + sameColorCounts) \* (k-1);
    3. sameColorCounts = temp;
* **House Robber II** (circular arranged houses)
* When it comes to DP but with circle, we use 2 sub problems to cover all situations: (1) not rob last house (2) not rob first house. And we’re done!
* **Stock buy and sell**
* only 1 transaction: easy, one iteration, get max diff
* unlimited transaction: easy, just add all increasing which prices[i+1]>prices[i]
* only 2 transaction: getMaxOneTransa(0,i) and getMaxOneTrans(i, length-1), then put together.
* With cooldown: use state machine model

****

* **Space optimize**:
* when n size problem only related to constant sub-problem, so use dp[constant] rather than dp[n]
* For example,

“House robber” only related to situation: rob/not rob last house;

“Paint color” only related to situation: paint same Color/different Color;

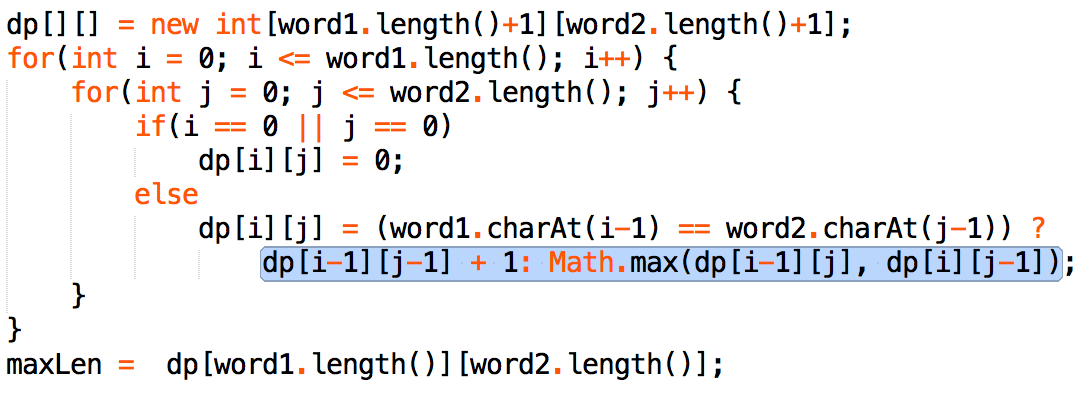
“Paint color with different price”: paint red/green/blue color;

we can use dp[2]; dp[2] and dp[3]

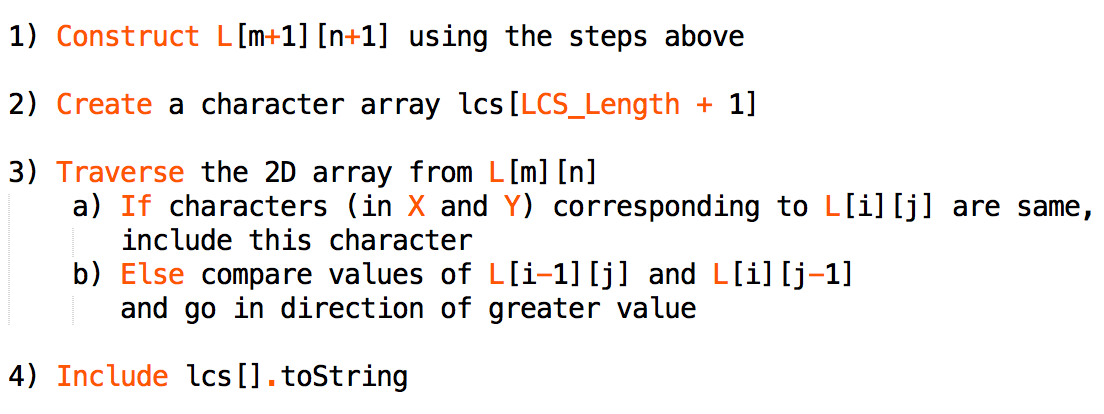
* To achieve circular buffer, we use **index % mod**

dp[i%2] = Math.max(dp[i-2]%2 + nums[i], dp[(i-1)%2]);

* **Longest Common Sequence**
* Get Maximum Length of LCS

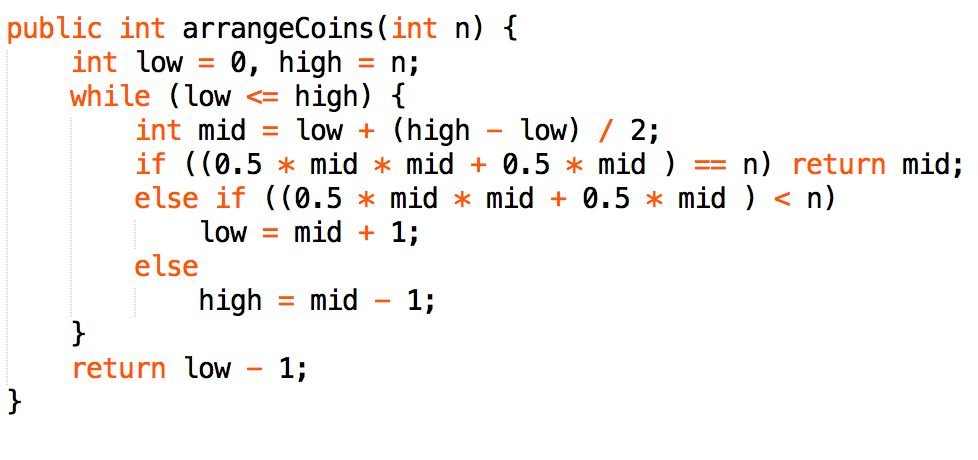
(https://www.programcreek.com/2014/04/longest-common-subsequence-java/)

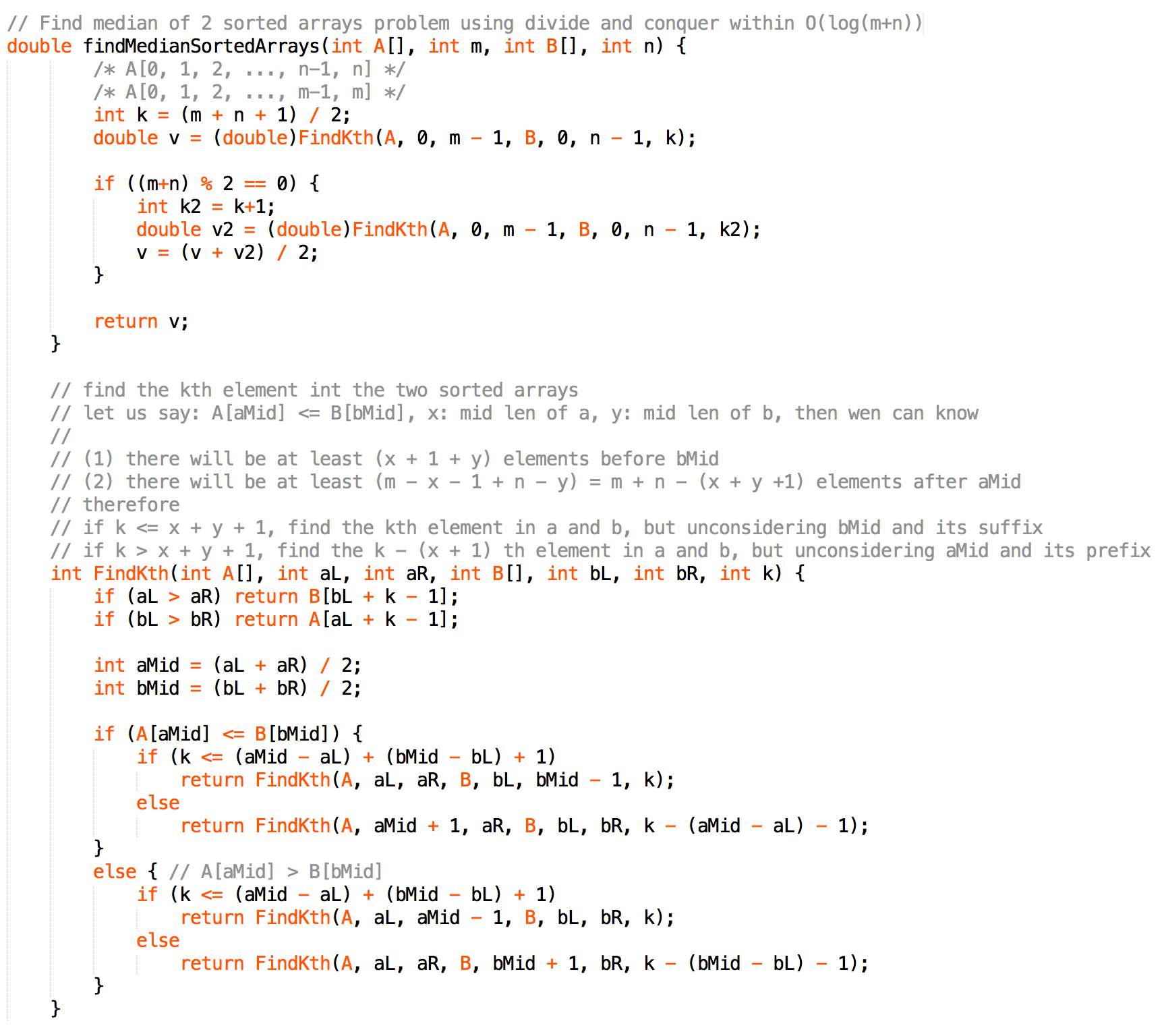
* Get String of LCS



* **DP relation glance:**
* Largest divisible subset (every pair (Si, Sj) of elements in this subset satisfies: Si % Sj = 0 or Sj % Si = 0.
* DP initiate mistakes 1. miss base case condition 2. problem lead by not fill dp[]

1. **Binary Search:**

* Arranging coins
* (x \* ( x + 1)) / 2 <= n; typical math + binary search problem
* binary search always have 3 problem of (1) low <= high / low < high (2) return low / return low – 1 (3) is when == , return mid or not
* (1) if definitely could find, < is enough. O/w, use <=
* (2) simply try some sample to decide
* (3) for edge detect problem, don’t return when equal (etc. First Bad Version
* 
* **Heaters Problem**
* Like meeting point problem, but try to find the minimum warm radius
* First sort(), then use Arrays.binarySearch() to find insert place of house
* Calculate Math.min(left dist, right dist(from heater)) iteratively, and finally get max.
* **Kth Smallest Element in BST**
* Easy solve in O(n) with in-order traversal, while another way is binary search which use countNode(Node) < = > k as condition statement
* **Kth smallst number of 2 sorted array**



1. **Divided and Conquer**

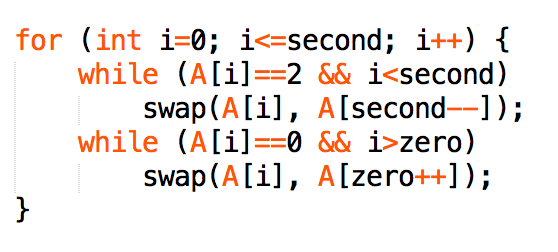
* **Different Ways to Add Parentheses**
* Input: "2-1-1". Since ((2-1)-1) = 0; (2-(1-1)) = 2; Output: List [0, 2]
* Since parentheses will affect both left and right side, DP is not available here
* So the right way is to divide string at each operator into substring(0, i) and substring(i + 1)
* recursively call diffWaysToCompute(String input) to get a List.
* Use nested for loop to sum up all combination division

1. **Stack**

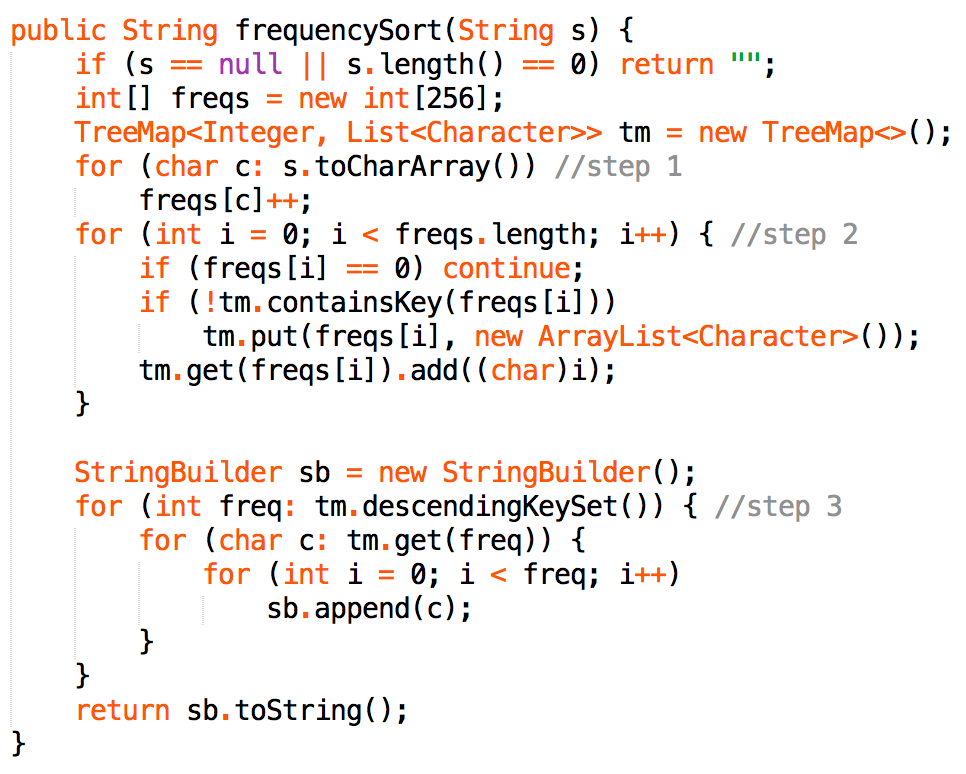
* **When Alignment from back (array | string | linked list), consider use stack**
* **Iterative Tree traversal problem**
* **Encode Decode problem**
* s = "3[a2[c]]", return "accaccacc"

1. **Heap**

* **Merge K Sorted Array**
* If there is only 2/3 array, we can assign pointers to each of them
* for more arrays, priority queue is needed to keep track of head of arrays
* **Sort Array**
* **Move Zero** (2 pointer; O(n) time; O(1) space)
* Just put non-zero element on the left, and complement 0 at tail
* **Sort Color** (2 pointer; O(n) time; O(1) space)
* like [1,0,1,2,2,1]
* must <= 3 colors, o/w use Arrays.sort() instead
* The idea is to sweep all 0s to the left and all 2s to the right



* **Sort Character by Frequency** (pq O(nlogn) time; bucket sort O(nlogn) time)
* Always use a int[256] ASCII to count character rather than map
* put each character into right bucket (TreeMap<Freq, List<character>>)
* Then we go through the bucket to get the most frequently character and append that to the final string builder

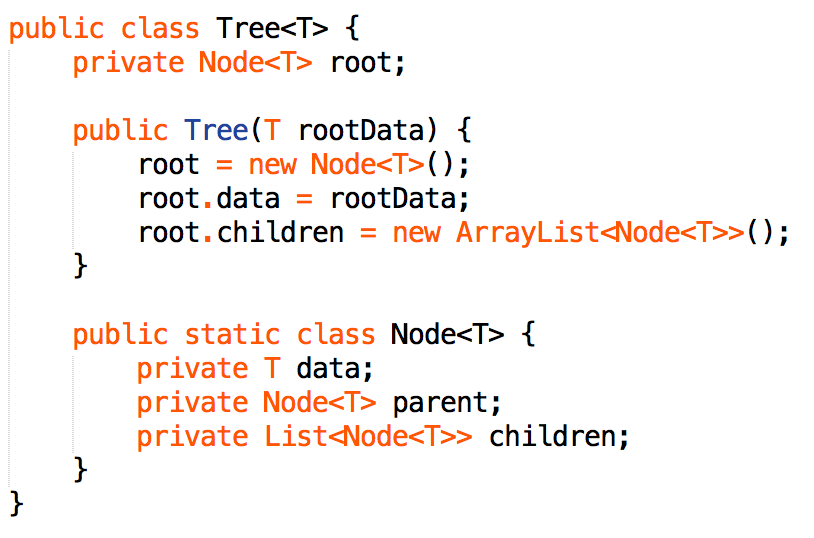


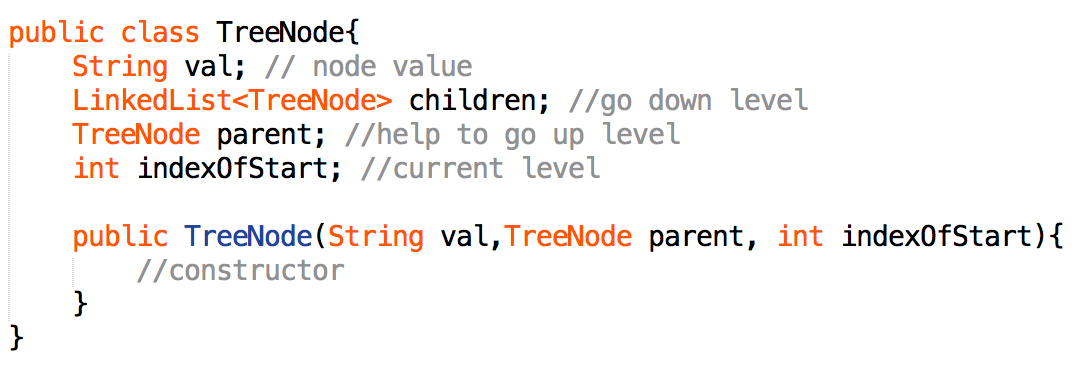
1. **Greedy**

* **Requirement for Meeting Room**
* First, sort interval by start time
* Second, use a min heap to track the minimum end time of merged intervals
* For each interval:
  + if the current meeting starts right after, there's no need for a new room, merge the interval
  + o/w, need a new room
* for all heap problem, always remember to put the ‘pop out’ back

1. **Tree**

* **4 common solutions**
* DFS
* BFS
* Up-down recursion call on 2 child node
* bottom-up recursive call, etc. Tree node-height problem
* “recursively call on 2 child node” is bound to application stack, cause less scalable and robust. And it could be replaced by iterative solution using stack.
* **Basic tree structure** (can be used for String or any other object)



* Design a directory
* Implement use tree structure
* Path Tree Sum
* PTS I (from root to leaf, return pathCount)
* Barely recursively subtract the value of current node from sum until it reaches a leaf node and the subtraction equals 0
* PTS II (root to leaf, return pathList)
* Use helper(). Keep a List<List<>> result and a List<> tmpLst to record current path List
* Don’t forget to remove last element of tmpLst after recursive call on both child node
* PTS III (from to any node, return pathCount)
* The key is to keep track of all possible current sum
* Use helper(). Keep a Map<Sum, Freq> map,
* if (map.containsKey(target – root.val))

count += map.get(target – root.val);

1. **Bit Operation**

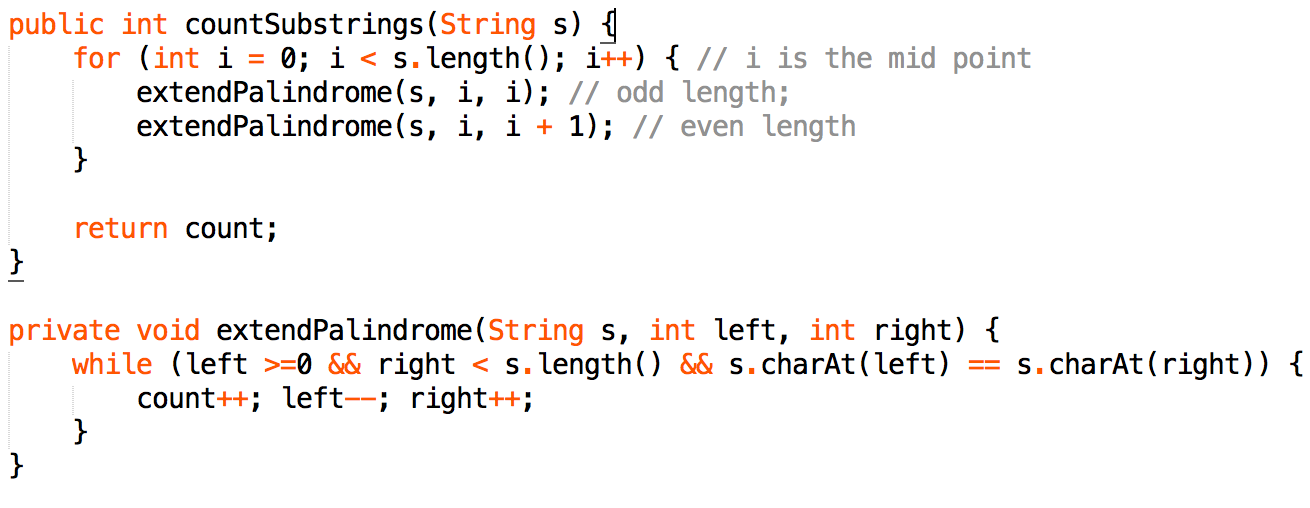
* Set union A | B
* Set intersection A & B
* Set bit A |= 1 << bit
* Clear bit A &= ~(1 << bit)
* Test bit (A & 1 << bit) != 0
* Extract last bit A&~(A-1)
* Remove last bit A&(A-1)
* considering single unique string or num, could use XOR
* Integer.bitCount(int) can be very useful in bit problem
* Binary addition, good to use for loop to decide finish:

for(int i = num1.length() - 1, j = num2.length() - 1; i >= 0 || j >= 0 || carry == 1; i--, j--){

Number base switch: n base to decimal: result = result \* 26 + (s.charAt(i) - 'A' + 1) or decimal to hexdecimal: num & 15

1. **Two Pointer**

* **Substring Problem**
* General way solve all substring problems by using hash map and counter within O(n) time
* problems including: Longest Substring with At Most Two Distinct Characters/ Without Repeating Characters/ Contains All Characters
* Palindromic Substring is an exception, which checks only new palindrome with length = current length +2 or +1



* **Trapping Rain water**
* Typical 2 pointer problem;
* if (leftmax < rightmax) {

(leftmax-A[a]) water can be stored; a++;

}

* **2Sum:**
* if already sorted, use 2 pointer; O(n) time, O(1) space
* watch out 2 pointer may probably meet, usually add  while(start<end &&…
* if not sorted, instead use hashmap; O(n) time, O(n) space
* 3Sum just extend 2sum (2 pointer) and narrow down the range
* **3Sum:** (return List<List<Integer>>)
* O(n^2) time is the optimal, so sort it and use 2 pointer, not hash map
* Remember to remove duplicate

**while** (lo < hi && num[lo] == num[lo+1]) lo++;

**while** (lo < hi && num[hi] == num[hi-1]) hi--;

* **4Sum:**
* using sub-functions for 3sum and 2sum, and keeping throwing all impossible cases.

1. **String**

* String.format("%d:%02d", h, m) is useful
* Character.isLetterOrDigit/isWhiteSpace/isUpperCase/isLowerCase(char c)
* Remove alphabetic letter: str.replaceAll(“[^a-Za-Z]”, “”);
* Remove numeric letter: str.replaceAll(“[^\\d.]”, “”);

1. **System Design**

* **Twitter Design**
* When need to return latest message from several users, create User and Tweet class
* User class has followedSet<Friend\_Id> and the head(latest posted) Tweet list.
* Tweet class is a Node to hold timestamp and a pointer point to next Node
* Whenever want to get most recent post of followers, use PriorityQueue<Tweet Head>
* This is the most efficient way, and use tweet head rather than post list will save space.

1. **Other**

* no extra space allow: try overwrite original data
* idea 1: move 0 to tail or remove sub-element problem)
* idea 2: mark index negative (when all the element in array are positive)
* check boundary: when go through a array or matrix, always remember to check the edge: i < grid.length
* String.substring(): str.substring(0, str.length()) = str.substring(0) = str
* Overflow: reverse a number (123 -> 321) or some other operation will cause overflow, how to detect and avoid?
  1. check calculate back to the x + y = z, and z - y = x? 2.
  2. Define long sum, when it is larger than the bound, then means overflow

**Appendix**

**1.Find Kth Smallest Number**

* Merge partition (no need to sort)
* T(n) = T(n/2) + O(n) = O(n)

