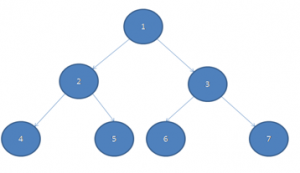
Amazon interview Questions:

1. Zig-zag traversal of a binary tree:  
   Given a binary tree as:  
     
   Make an algo that prints 1 3 2 4 5 6 7 (and more)  
   **Solution:** use a queue and a stack. Perform a level-order traversal using the Queue (condition: while queue is not empty). Keep track of the level by using a wrapper structure.   
   Assume the level of the root is 1.  
   If the current level is an odd number, just print when you dequeue and don’t use the stack.   
   If the level is an even number, push onto the stack when you dequeue. Then, once the level *changes*, pop and print all from the stack. Doing so ensures that every even-numbered level gets printed in the reverse order it would during regular level-order traversal. You might need to pop all from the stack after the loop.  
   Code: [in C++](https://github.com/ARDivekar/Algorithms/blob/master/Interview%20Practice/Amazon/zigzag%20Binary%20Tree%20traversal.cpp)  
   It’s also possible to do it recursively: [GeeksForGeeks](http://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/) but this is O(n^2) time for skewed trees with O(h) stack space (which becomes O(n) if the tree is skewed). The above method is O(n) time with O(n) space.  
   It’s also possible to do this with a special kind of queue that, on demand, goes backwards once completely (thus serving the purpose of a stack; we pop from the queue and push back into it, for each element in the queue).
2. Rotate a matrix by 90 degrees: this is a common question, and it’s also in Cracking the Coding Interview.  
   Basically, if it’s a square matrix, you do this: rotate the outermost ring first (i.e. the four overlapping arrays on the edges) by 90 degrees. Then you recursively do the same for the inner rings, from the outside in.   
   For an mxn matrix, if m!=n, then there’s a problem of actual space: you have to store it in a completely different set of arrays. So, might as well just copy it.
3. Rotate k alternate nodes of a linked list:  
   Example1:  
   Inputs: 1->2->3->4->5->6->7->8->NULL and k = 3   
   Output: 3->2->1->6->5->4->8->7->NULL.   
   Example2:  
   Inputs: 1->2->3->4->5->6->7->8->NULL and k = 5  
   Output: 5->4->3->2->1->8->7->6->NULL.  
     
   **Solution:** The simplest solution is disconnect every k nodes, pass it to a function that reverses linked lists, then reconnect it. This basically goes over the list twice. It requires constant auxiliary space (not even for the call stack). The time is O(n).  
   An iterative solution can also be done by reversing as we traverse the list. We keep a temporary head pointer for the local head, and effectively reverse the first ‘k’ nodes of the linked list starting from that head. The same can be done recursively, and is implemented [here](http://www.geeksforgeeks.org/reverse-a-list-in-groups-of-given-size/).  
   Code: in C++
4. Link nodes of a tree which are at the same level:  
   Input Tree

A

/ \

B C

/ \ \

D E F

Output Tree

A--->NULL

/ \

B-->C-->NULL

/ \ \

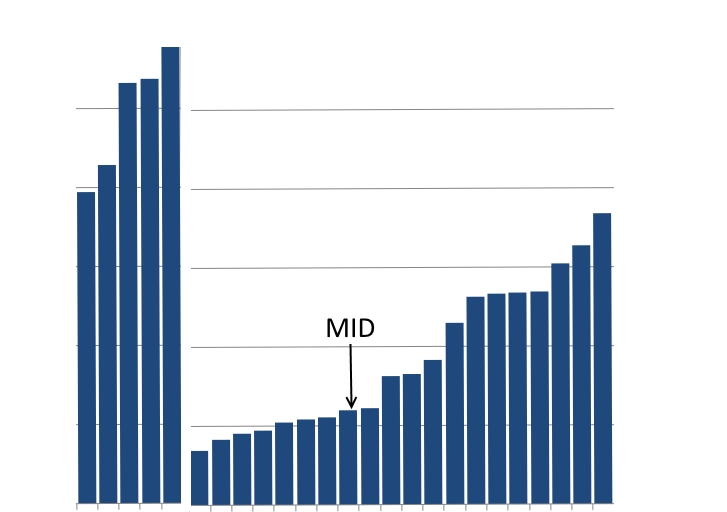
D-->E-->F-->NULL  
  
Solution: you might think level-order traversal, and you’d be thinking in the right direction. Only, we don’t need a queue: you can re-use the linked-list made on the previous level to get the nodes on the next level. This makes it constant space, discounting the space needed for the pointers. Honestly, just create a linked-list for each level, it’s a lot easier to manage; just create a wrapper structure with two parts: a pointer to the TreeNode object, and the pointer to the next TreeNode object (sideways).   
You can do it without a linked list in proper O(1) space if your nodes already have the provision for a nextRight pointer which can be made to point to a sibling node.  
Source of solution: Cracking the Coding Interview.  
You can also do this recursively by having a global variable of the array of linked lists for each level, and when you get to each level (pass the level to the recursive method) just append to that linked list of that level.

1. Find the first non-repeating character in a string:  
   This is simple if we have a constant character range (it may be a very large constant, like Unicode, but constant nonetheless): just use a Hashmap of the counts of each character. To get the *first* non-repeating character, use a doubly linked list that links all the characters as they are seen. Eg: “this is a call” will be linked as “t->h->i->s->a>c>l”. We only add a new node at the end of the LL if a character has not been seen before (verified with the hashmap). At the end, we go over the linked list once, from the head, and eliminate nodes which have characters that have counts more than one. So, in our pruned linked list, we only have characters which appear once, in chronological order. Since we want the first non-repeating, we can return the head, but we can also get the last, second-last etc. with this method. This entire solution is O(N) time, where N is the length of our string.   
     
   This problem can be extended by saying that we have an infinite stream of characters. The above solution doesn’t exactly work for this case, as there is no ‘stop’ point after which we prune our linked list. So, we prune on the go: if the character in the head node appears for the second time, then we remove it and go to the next. We check the next, to verify that it appears only once. If it does not, we remove it too. So on, until we reach a non-repeating character or have an empty list. The good thing about this method is that since a node removed from the list cannot enter back into it at some later point, we will have at most |characters| number of nodes to prune, which is constant.  
   Note: for both of the above, it is not absolutely necessary to use a hash table as an array; we can save a lot of space if we just use a bit vector (essentially an integer) that tells us if the element has been seen before or not. This cuts down our space by a lot.   
     
   Another extension of the problem is when we have a non-constant number of characters, but still want to have constant time i.e. O(1). Luckily, finding the first non-repeating one is the part of the problem that does not change. What changes is how we use our hash table. Having a non-constant number of characters means that we will probably have to use something like linear probing with table doubling, or make linked lists of the hash table. I prefer the first method (linear probing) as it uses less space (no pointers), and because the linked list-hash table will get very slow if we grossly underestimate the number of unique characters we have. Table doubling ensures that linear probing is fast on average, even if it is expensive on occasion, even if we have a LOT of characters.
2. Longest palindromic substring: find the longest palindrome (i.e. same backwards and forwards) in a string.  
   **Solution:** via geeks for geeks in O(n^2) time but O(1) space: consider each index of the array to be the centre of the palindrome. So, starting at that index, grow to the left and the right one character by one, until it is no longer a palindrome or you hit an edge of the string. Store the start and end points of the longest palindrome with that index as the centre. Do this for each index.   
   Note: when checking the centre points, there are two cases: an even-numbered palindrome and an odd-numbered one. The even-numbered one requires we take that as the centre index, and then one before it, check if they are the same character, and *then* grow. Else, we use the odd-length palindrome, where we just grow from the index.   
   Side note: maybe something like Rabin-Karp will let you do this faster.
3. Special stack: push, pop and min in O(1): push and pop are easy. Min is the problem: we want to *repeatedly* be able to get the min in O(1) time, while also maintaining the O(1) time for push and pop. This makes things tricky. But understand one thing: we can only access the top of the stack. So, we don’t need to save the minimum of all the elements: we just need to save the minimum for all the elements *currently in the stack*.  
   So, we use two stacks: one for the actual data and one auxiliary stack, which, stores the minimum seen **so far**. So, (assuming we append at the end) Aux[i] stores the minimum of stack S[0…i]. If S[i] had been greater than S[i-1], Aux[i]=Aux[i-1],i.e. we store repeat minimums in the Auxiliary stack. Thus, it stores the *minimum seen so far*. [For more detail](http://www.geeksforgeeks.org/design-and-implement-special-stack-data-structure/).  
   We can space-optimize the Auxiliary stack by compressing the repeated minimums into only one, and when popping from the special stack, check whether to pop from the compressed auxiliary stack also.  
   Extending the problem: median also.  
   Solution: getting the median in O(1) is even more tricky. One way is to use two heaps, a min-heap and a max-heap, to dynamically maintain the median. Pushing thus becomes O(lgN), and popping become O(N) as we need to search the heaps, but median becomes O(1). Alternatively, while popping, you might need a hashtable, which, given the top of the stack, can tell you it’s location in the heap, so that we can remove that element from the heap.
4. Left view of a Binary Tree: a level order traversal wherein you only print the first (leftmost) node on each level.   
   Similarly, right view of tree.  
   Doing this with level-order is easy enough. We can also do this recursively with an inorder traversal: since we recursively go left in an inorder traversal, we keep track of the lowest level we have ‘handled’ so far. If the node we are on is at a lower level, we print it and reset the lowest level we have handled so far.   
   On [geeksforgeeks](http://www.geeksforgeeks.org/print-left-view-binary-tree/).
5. Given two linked lists of digits to represent numbers, make a linked list that is the sum of the two numbers.  
   Eg: 1 -> 5 -> 3 -> 4 and 8->3   
   print 1 - > 6 -> 1 -> 7  
   Solution: This is simple in one direction, but not as simple in the other direction. Which direction is it simple? The one where the carry must propagate towards the end of the list, eg: 2->5->3->4 + 8->3 = 0->9->3->4 (i.e. 4352+38)  
   What about the not-simple type, like in the given example? It has a few problems, like we don’t know when to start traversing the smaller list until we know both the lengths. One easy fix is to just reverse both and apply the function we made for the simple one, then reverse both again.   
   Otherwise, if the lengths are given, then at the appropriate time start traversing the digits of both lists simultaneously. Keep a pointer back to the previous node, for the carry. Even if the lengths are not given, use this method because the reversing method is three list traversals, while this is only two.
6. Find the point of looping in a linked list:   
   Solution:  
   The most straightforward implementation is with a hashtable that stores the addresses of the nodes. That’s O(N) space.   
   It’s possible to do it in O(1) space using the slow pointer-fast pointer method (this is called Floyd’s cycle): from the head of the LL, start a slow pointer and a fast pointer (which is 2x the slow pointer). Go until they meet (they will). From the meet point and from the head, start two slow pointers. The point where they meet is the start of the loop. You can prove all this mathematically quite simply.
7. Given a BST, replace each node with the sum of the values of all the nodes that are greater than that node. Only constraint being that you are not allowed to use any global or static variable.  
   Solution: This may seem complicated at first, but when you think about the inorder traversal (which is sorted), it becomes much simpler.   
   So, all we have to do is find the sum of all the nodes in the tree (which can be done with any traversal). Then, going by inorder, starting with the smallest node, we must get do sum-(sum of the nodes seen so far). Replace the data of the node with this value.   
   for a sorted array: [1, 3, 6, 9, 10, 11], our sum is (1+3+6+9+10+11) = 40, and our output array would be [40-(1), 40-(1+3), 40-(1+3+6), 40-(1+3+6+9), 40-(1+3+6+9+10), 40-(1+3+6+9+10+11) ].
8. Given an array of numbers find a triplet that satisfies the given condition.   
   Condition: a[i] < a[j] < a[k] where i < j < k.   
   If there are more than one triples, print them all.  
   [Solution:](http://www.geeksforgeeks.org/find-a-sorted-subsequence-of-size-3-in-linear-time/)  
   This question has a rather surprising conundrum. Ask the interviewer to clarify what s/he means by ‘all’ triples.  
   For a given element at index ‘i’ in the array A[0…n-1], it is possible to know if a triple *exists* with that element as the central element, in O(n) time, for all triples. However getting *all* the triples that exist is an O(n^2) task, and generating them is an O(n^3) task (because, there can be at most O(n^3) triples, for a sorted array of unique elements).  
   The solutions for both cases follow the same pattern: maintain arrays called smaller[0…n-1] and greater[0…n-1]. These are assumed to be initialized to -1. If there exists an element A[i] that has an element smaller than it in A[0…i-1], then we set smaller[i] as the index of that element. Similarly, we set greater[i] to be the index of an element, in A[i+1…n-1], if it exists. Then, we iterate through A[i] a second time, and if smaller[i]!=-1 and greater[i]!=-1, we print the triple( A[ smaller[i] ] , A[i], A[ greater[i] ] ).   
   So, how do we do this? The answer: we store the min we have seen so far, to set smaller. Concretely:  
   min\_so\_far=a[0]  
   for(i=0; i < n; i++)  
    if(a[i] <= arr[min\_so\_far]):  
    min\_so\_far=i  
    smaller[i]=-1  
    else: smaller[i]=min

We do a similar thing with the max\_so\_far to set greater, only we start from the other end of the array. Then, we do a final, third traversal to print the triples. The runtime is O(n) with O(n) auxiliary space.  
  
Extending this problem, we may be asked to print **all** triples that exist. There are O(n^3) such triples: for a sorted array of unique elements, a every element with the set of all elements before it, and every element with the set of all elements after it, forms a triple. Just printing all these triples would take O(n^3) time. The funny thing is, we can store all of them in O(n^2) time and O(n^2) space: instead of an array of integers of smaller[i], use an array of hash tables (hash tables to avoid repeat values). Do the same for greater[i]. Now, go over the array exactly O(N^2) times in two nested loops, and if A[j] < A[i] && j<i, append j to smaller[i]. If j>i && A[i]<A[j], append j to greater[i]. Then, for each i, loop through the left and right hash tables (nested) and print all triples. This printing step is O(n^3).

1. Given an array where we have all elements occurring twice except for a particular element which occurs once, find that element.  
   You might think to hash, but you can actually do this in O(1) space, using the XOR function  
   eg: int x = 5^3; //x==6 as (101 XOR 011) = 110 = 6  
   We should remember three properties of XOR:
   1. XORing a number with itself gives us 0000
   2. XORing a number with 0000 gives us the same number.
   3. The order of XORing does not matter.
   4. Note: additionally, XORing a number with a list of 1’s of the same length, complements the number.

These two properties allow us to solve the problem in O(1) space: we just iterate through the input array, keeping a temp variable starting as 0. We do (temp ^ a[i]). The order of XORing does not matter. So, this variable acts as a sieve, allowing the repeated elements to pair off with each other and cancel out. The only remaining number is the odd one out, which occurs only once.   
This method works only if there is at most one number that repeats an odd number of times, and the others all repeat an even number of times (zero is even).

1. Given a sorted array of 0s and 1s find the point of transition from 0 to 1:  
   This might seem silly, but the trick if you should use a modified binary search and do it in O(lg N), since the array is sorted.
2. Find an element in a sorted array which has been rotated an unknown number of times to the left or to the right:  
   First, notice that there is no such thing as a left or right rotation: a left rotation by k is just a right rotation by N-k. So, consider that all rotations are done towards the left. Second, this is a sorted array, so we need to modify binary search and get an O(lg N) algo.  
   What we are looking for is the ‘cliff’ point: where the value suddenly drops from high to low. We need to know, at any point in time, if the cliff point is to the left of the ‘mid’ or the right. How do we do that?   
   Look at A[low], A[mid] and A[high]. Start with low=0, high=A.length, mid =(low+high)/2.  
   For all rotations, A[0] > A[N-1]. The question is, is A[low] > A[mid]? If so, the cliff point lies to the left. Otherwise, if A[mid] > A[low], then the cliff point lie to the right.   
   A graphical example might help here:  
     
   Take low=0. If A[low] > A[mid] (as it is here), then the cliff point is to the left of the mid. Otherwise, if A[low] < A[mid], the cliff point (i.e. the end of the not-rotated array) is to the right.  
   Once we have the branching condition, we’re done. Just keep using it until we find that A[mid-1] > A[mid].   
   The next half of the solution, the actual searching for the element, is another modified binary search: start at low = cliff point index, & high = cliff point index + N. ‘Map’ to the array using (index)%N, but maintain low, mid and high between cpi & cpi+N.
3. Given a ‘mountain’ array, find an element: (note: one is ascending and the other is descending).  
   Solution: Since it is find and the arrays are sorted, you must modify the binary search algo and run in O(lg N).  
   What you do is this: Like normal binary search, have low=0 & high=A.length. Start at the mid=(low+high)/2. If A[mid-1] < A[mid] < A[mid+1], you know that it is increasing, so the peak must lie to the right, make low=mid+1. If A[mid-1] > A[mid] > A[mid+1], branch to the right. Continue until A[mid-1] < A[mid] > A[mid+1]. So now, mid is the peak.   
   Then, just use normal binary search on the arrays to find the element.  
   Note: one of the tricky things to handle is the plateau values: if A[mid-1] == A[mid] == A[mid+1]. This is not a problem if all the elements are unique.
4. Convert a BST to a LL in-place:  
   1. Inorder LL
   2. Preorder LL
   3. Postorder LL

Ans: We need to convert a BST node into a doubly linked list node, and for that we have the following convention:   
BSTNode->right becomes Node->next   
BSTNode->left becomes Node->prev, and   
BSTNode->parent becomes NULL, or is unaffected.  
Following this convention, we can perform all the traversals as normal in their recursive implementation, with one change: in every ‘visit node’ operation, we perform a function called ‘remove\_and\_return’, which performs the procedure to delete a node from a BST by replacing with left child/right child/successor, and then – instead of deallocating the node – appending the node to the end of the linked list. This process is necessary because performing remove() does not change the BST ordering for inorder, postorder, or preorder.

1. Get the kth largest element in a BST:  
   The naïve solution is to use inorder traversal, which is O(n). And indeed, if we are just presented with a BST, that’s the only way to do it.  
   However, if we are allowed to monitor the BST while it is being built, then while inserting we can store the count of elements in the left and right subtree of each node (while going downwards, if we take a left branch, increment the count of the left subtree for that parent node. Do this for all nodes on the path).   
   So now, suppose we are given a BST with the counts of the number of nodes in the left and right subtree. All we have to do is use a sort of Binary search:  
   Assume we have to find the Kth largest element. Assume that the root has N nodes in its left subtree. If K = N + 1, root is K-th node. If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element. For a balanced binary tree, this will be like binary search, in O(lg N) time. For a very skewed tree, it will be O(N). Note that we technically need the count of elements in left subtree only.
2. Given two sorted arrays, get the median of their merge in O(lgN) time.  
   As you may expect, we don’t actually merge them as that is O(N). We have to reduce the problem in half at every step to make it O(lgN).   
   The idea is this: start with the medians of the two arrays. If they are equal, that must be your final median. If not, see which is greater than the other.  
   eg: A={1,2,3,6,8}, B={2,3,5,7,9,10} . A[mid] = 3, B[mid]=7. 3 != 7, so the median must lie between 3 and 7 in the merge of the two. Thus, eliminate the case of the left half of A (less than 3) and the right half of B (greater than 7). Continue the procedure for the now-smaller arrays. If A[0….N-1] and B[0…M-1], this take O(lg(N+M)) time.  
   The real tricky part is the base cases: there are six of them which we must handle:
   1. N=1, M=1 : return the average of the two.  
        
      For the rest of the cases, assume N<=M (we can swap arrays to make this the case)
   2. N=1, M=odd:   
      we consider mid=M/2, i.e. M=7, mid=7/2 = 3 <-index
      1. If A[0] is less than B[mid] and B[mid-1] then return avg(B[mid], B[mid-1])  
         eg: B = {5, 10, 12, 15, 20, 25, 30}, A={6}
      2. If A[0] is greater than B[mid] and B[mid+1], return avg(B[mid], B[mid+1])
      3. If B[mid-1] <= A[0] <= B[mid] OR If B[mid] <= A[0] <= B[mid+1],   
         return avg(A[0], B[mid])  
         eg: B = {5, 10, 12, 15, 20, 25, 30}, A[0] = 13 OR A[0]=17, return avg(A[0], B[mid]
   3. N=1, M=even:  
      eg: B={1,3,4,6,7,8}, A={4}  
      Here, we need the middle two elements, B[mid-1]=4 and B[mid]=6, where mid=M/2 = 6/2, = 3 <-index
      1. If A[0] <= B[mid-1], return B[mid-1]   
         eg: A[0]=4, return 4
      2. If A[0] >= B[mid], return B[mid]   
         eg: A[0]=9, return 6
      3. If B[mid-1] <= A[0] <= B[mid], return A[0]
   4. N=2, M=odd:   
      **median** = median of elements (B[mid] , max(A[0], B[mid-1]), min(A[1], B[mid+1]) )  
      Where mid = M/2 (rounded down)  
        
      eg: A={2,3}, B={1,4,7,8,10}.   
      M=5, mid=2  
      Median = median of (7, max(2,4), min(3,8)) = median of (7, 4, 3) = 4.  
      Merged: 1,2,3,**4**,7,8,10  
        
      eg2: A={3,9}, B={1, 4, 6, 7, **10**, 10, 12, 13, 14}. M=9, mid=4 B[4]=10  
      Median = med of(10, max(3,7), min(9,10)) = med of (10,7,9) = 9  
      Merged: 1,3,4,6,7,**9**,10,10,12,13,14
   5. N=2, M=2: just find the median of 4 elements
   6. N=2, M=even:  
      **median** = median of ( B[mid], B[mid-1], max(A[0], B[mid -2]), min(A[1],B[mid+1]) )  
      Where mid=M/2 (rounded down)  
      eg: A={1,9}, B={2, 2, 6, 7, 10, 14}  
      M=6, mid=3 B[mid]=7.   
      The middle two elements are B[mid-1] and B[mid], so we are considering the middle *four* elements: B[mid-2], B[mid-1], B[mid], B[mid+1]  
      median = med of (7, 6, max(1,2), min(9,10)) = med of (7, 6, 2, 9) = avg(6,7) = 6.5

The code for all these cases is quite long, so it might help to have the helper methods min(a,b), max(a,b), average(a,b), median(a,b), median(a,b,c) and median(a,b,c,d).  
  
The rest of the code is just the narrowing down to these cases for larger arrays. We do so recursively by taking the medians of A and B, and eliminating the halves we don’t want, then passing it to the same function.   
  
We can make a small optimization if one array is completely less than the other: find the median directly.  
eg: A={8,9,10,15}, B={1,2,4,7}, B[M-1] <= A[0], so median is (N+M)/2. If M-1 < (N+M)/2, median = A[ ((N+M)/2) % M ] (or something).

1. Given an array of numbers, check if it is possible, with additions and subtractions placed anywhere, to make the sum zero.  
   eg: A={2,1,8,5}  
   +2-1+8+5 != 0  
   -2-1+8+5 != 0  
   +2-1-8+5 != 0  
   -2-1+8-5 == 0 thus done.  
     
   Solution: there is no solution in polynomial time, as this is the partition problem, which is NP-complete. In the partition problem, we must distribute a given set of numbers into two groups, such that their sum is the same. Considering the two groups here to be + and -, and the sum being the same means they add up to zero, it is the partition problem.
2. Given an integer, determine all the possible words you can make on a phone keypad by entering those integers.  
     
     
   For example if input number is 234, possible words which can be formed are (Alphabetical order):

adg adh adi aeg aeh aei afg afh afi bdg bdh bdi beg beh bei bfg bfh bfi cdg cdh cdi ceg ceh cei cfg cfh cfi (27 in total)  
Solution: the mapping of numbers to alphabets is trivial, they just want you to figure out how to print all the possible combinations. The problem is that you don’t know how long the input integer is. So, you can’t use a solution that directly uses loops to print, because you can’t have a dynamic *number* of loops.  
However, there is a pattern, and it lets us print in sorted order:  
If there are N digits in the string, there are exactly 3^N different possible strings.   
Imagine, instead of building the strings one string at a time, we build them in parallel:   
input = 234, where 2={a,b,c} ; 3={d,e,f} ; 4={g,h,i}  
Consider these above 27 inputs, stored in an array of strings: (N=3; thus 3^3 =27)  
[0] a d g  
[1] a d h  
[2] a d i  
[3] a e g  
[4] a e h  
.  
.  
[7] a f h  
[8] a f i  
[9] b d g  
.  
.  
[17] b f i  
[18] c d g  
.  
.  
[26] c f i  
We can build the array in parallel: for the first 9, i.e. 3^(N-1), we only use ‘a’ as the first letter. The next 9, the first letter is only b. Last 9, first letter is only c.   
Consider the 2nd letter in all strings. It changes every 3 strings i.e. 3^(N-2), then loops back. The 3rd letter changes every string, i.e. every 1 time, i.e. every 3^(N-3) times. You can see the pattern.  
You can use this pattern to print every possible string without having to build it in parallel. Just keep a track of the current number of the string you are printing (i.e. 0 to 26), and use that to calculate which digit to use for each input number in the integer, while printing.   
So, we use two loops: one loops through strings and one loops through the input integer multiple times.   
Time complexity is O(N . 3^N), because we iterate through the integer for each string to determine which alphabet to print, but that’s unavoidable.

1. Find the 2nd -most repeating element in a sorted array, in O(1) space:  
   We sort it, and count the first and second most repeating using the [Boyer–Moore majority vote algorithm](https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_majority_vote_algorithm). Basically, we keep four variables: max\_elem, max\_count, second\_max\_elem and second\_max\_count. Since the array is sorted, once we go over a block, we can tell if we need to update either of the two pairs of variables by comparing the count of the element in that block.   
   For a general kth most repeating elements, we need O(k) space.   
   We can use the same solution for the kth *least* repeating element (discounting those which do not occur at all).
2. Find the number just greater than a given number, with the same digits.  
   Eg: input num = 22034, just greater = 22043  
   Soln:   
   Notice that the number 9876 or 7432 or any number with the digits in decreasing order, has no greater number. Why? Because it has no *reversal*, i.e. where the number on the left is less than the number on the right, if we start from the rightmost digit of the number.   
   Also notice that for 745, the greater number is 754, because ‘45’ is the first reversal. Similarly, 7772543’s greater is 7773245, because ‘25’ is the first reversal. Essentially, the numbers on the left of the reversal are unaffected.  
   So, we only have to check pairs of numbers from the right, up to the first reversal. What next? Well, we find the smallest digit on the right of the reversal which is greater than the left digit of the reversal (in the worst case, this will be the right digit of the reversal) and we put it in the place of the left digit of the reversal.   
   Then, taking the rest of the numbers we have seen so far (including the left digit of the reversal), we sort them and append them to the rest of the number.   
   i.e. number = 77748932   
   left digit of reversal = 4  
   number just greater than 4 = 8   
   put 8 in the position where 4 was, i.e. 777----- -> 7778----  
   put the other numbers in sorted order, i.e. 77782349  
   We can do the sorting with counting sort, while we are going from right to left. Since there are a finite variety of digits, this takes O(1) space and O(n) time, for an n-digit number.  
   [C++ code](https://github.com/ARDivekar/Algorithms/blob/master/Interview%20Practice/Amazon/number%20greater%20than%20given%20with%20%20same%20digits.cpp).  
   Extension: find the just-smaller number; this is the same problem, only we check for the opposite kind of ‘reversal’, where the left number is larger than the right. We still check from right to left, but the rest of the numbers we append in the reverse-sorted order.  
   Extension 2: do the same for a string; use same procedure. Use counting sort with an array of counts, int char\_counts[256]; for all the ASCII characters.