Contents

http://www.ardendertat.com/2012/01/09/programming-interview-questions/

Programming Interview Questions 1: Array Pair Sum

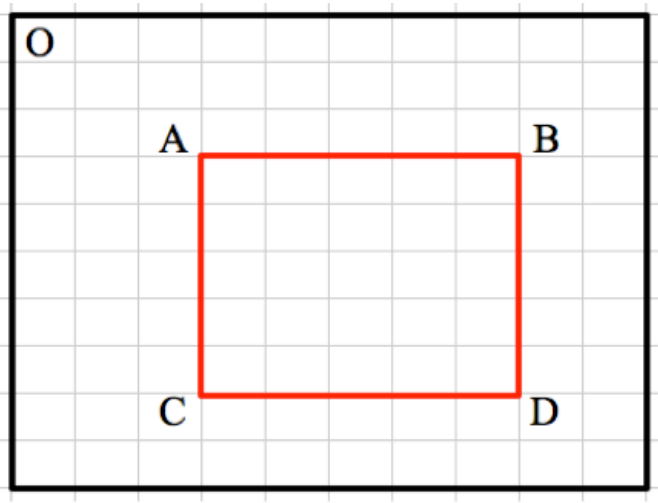
Given an integer array, output all pairs that sum up to a specific value k.

A more efficient solution would be to sort the array and having two pointers to scan the array from the beginning and the end at the same time. If the sum of the values in left and right pointers equals to k, we output the pair. If the sum is less than k then we advance the left pointer, else if the sum is greater than k we decrement the right pointer, until both pointers meet at some part of the array. The complexity of this solution is O(NlogN) due to sorting.

The O(N) algorithm uses the set data structure. We perform a linear pass from the beginning and for each element we check whether k - element is in the set of seen numbers. If it is, then we found a pair of sum k and add it to the output. If not, this element doesn't belong to a pair yet, and we add it to the set of seen elements. The algorithm is really simple once we figure out using a set. The complexity is O(N) because we do a single linear scan of the array, and for each element we just check whether the corresponding number to form a pair is in the set or add the current element to the set. Insert and find operations of a set are both average O(1), so the algorithm is O(N) in total.

Programming Interview Questions 2: Matrix Region Sum

Given a matrix of integers and coordinates of a rectangular region within the matrix, find the sum of numbers falling inside the rectangle. Our program will be called multiple times with different rectangular regions from the same matrix.



70	37	23	57	27	22	90	99	22	59
47	63	33	1	42	46	6	70	98	93
36	62	50	21	92	27	60	29	15	34
53	3	88	45	57	39	83	81	79	56
28	63	89	20	47	15	84	18	82	33
26	87	11	76	79	5	94	55	73	51
17	82	86	10	96	5	42	43	51	6
44	76	51	4	15	99	52	11	70	89
66	36	92	85	50	21	72	27	52	65
60	0	67	37	59	14	33	13	36	36

Programming Interview Questions 3: Largest Continuous Sum

Given an array of integers (positive and negative) find the largest continuous sum.

If the array is all positive, then the result is simply the sum of all numbers. The negative numbers in the array slightly complicate things. The algorithm is, we start summing up the numbers and store in a current sum variable. After adding each element, we check whether the current sum is larger than maximum sum encountered so far. If it is, we update the maximum sum. As long as the current sum is positive, we keep adding the numbers. When the current sum becomes negative, we start with a new current sum. Because a negative current sum will only decrease the sum of a future sequence. Note that we don't reset the current sum to 0 because the array can contain all negative integers. Then the result would be the largest negative number.

Kadane algorithm

Programming Interview Questions 4: Find Missing Element

Here is an array of non-negative integers. A second array is formed by shuffling the elements of the first array and deleting a random element. Given these two arrays, find which element is missing in the second array. Here is an example input, the first array is shuffled and the number 5 is removed to construct the second array.

First Array = [4, 1, 0, 2, 9, 6, 8, 7, 5, 3]

Second Array = [6, 4, 7, 2, 1, 0, 8, 3, 9]

A more efficient solution is to sort the first array, so while checking whether an element in the first array appears in the second, we can do binary search. But we should still be careful about duplicate elements. The complexity is O(NlogN).

If we don't want to deal with the special case of duplicate numbers, we can sort both arrays and iterate over them simultaneously. Once two iterators have different values we can stop. The value of the first iterator is the missing element. This solution is also O(NlogN).

We can use a hashtable and store the number of times each element appears in the second array. Then for each element in the first array we decrement its counter. Once hit an element with zero count that's the missing element. The time complexity is optimal O(N) but the space complexity is also O(N), because of the hashtable.

initialize a variable to 0, then XOR every element in the first and second arrays with that variable. In the end, the value of the variable is the result, missing element in array2.

```
    for num in array1+array2:
        result^=num
    return result
```

Programming Interview Questions 5: Linked List Remove Nodes

Given a linkedlist of integers and an integer value, delete every node of the linkedlist containing that value.

```
while (head!=NULL && head->val==rmv)
    {
        Node *temp=head;
        head=head->next;
        free(temp);
    }
    if (head==NULL)
        return;

    Node *current=head;
    while (current->next!=NULL)
    {
        if (current->next->val==rmv)
        {
            Node *temp=current->next;
            current->next=temp->next;
            free(temp);
        }
        else
        {
            current=current->next;
        }
    }
}
```

Programming Interview Questions 6: Combine Two Strings

We are given 3 strings: str1, str2, and str3. Str3 is said to be a shuffle of str1 and str2 if it can be formed by interleaving the characters of str1 and str2 in a way that maintains the left to right ordering of the characters from each string.

For example, given str1="abc" and str2="def", str3="dabecf" is a valid shuffle since it preserves the character ordering of the two strings. So, given these 3 strings write a function that detects whether str3 is a valid shuffle of str1 and str2.

We have to use recursion to solve the problem. Firstly check the length of the str equals the length of str1 plus the length of str2.

If the first characters of str1 and str3 are the same, then we'll recurse with new str1 and str3 being all but first characters of the strings, and str2 will stay the same.

If first characters of str2 and str3 are the same, then we'll do the same thing with new str2 and str3 being all but first characters, and str1 the same.

if neither str1's nor str2's first character equals str3's first character, we return false.

```
if str1[0]!=str3[0] and str2[0]!=str3[0]:
        return False
    if str1[0]==str3[0] and isShuffle(str1[1:], str2, str3[1:]):
            return True
    if str2[0]==str3[0] and isShuffle(str1, str2[1:], str3[1:]):
            return True
```

The algorithm effectively uses recursion to solve a smaller instance of the same problem. However, the complexity is exponential. Since we don't cache the evaluated results, we may try to evaluate the same input strings again and again. So, we need to perform dynamic programming and cache the already evaluated results to avoid precomputation.

```
def isShuffle2(str1, str2, str3, cache=set()):
    if (str1, str2) in cache:
        return False
if str1[0]==str3[0] and isShuffle2(str1[1:], str2, str3[1:],
cache):
            return True
    if str2[0]==str3[0] and isShuffle2(str1, str2[1:], str3[1:],
cache):
            return True
     cache.add( (str1, str2) )
```

```
    return False
```

The cache is a set where the key is the tuple of str1 and str2. We cache the values we already know that can't produce a valid shuffle and check before trying again. The complexity of this solution is O(NM) where the N and M are the lengths of str1 and str2 respectively. So, from exponential we reduced the complexity to quadratic by using dynamic programming. This is the worst case complexity though, average case would be better.

I wonder if it would be possible to keep only the index number of failing string compared to original one in the cache instead of whole substring. For your example, it would cache the string tuple as {bc,f}, for some time rigth? Instead, would it be possible to cache {1,2} meaning that 1 is the substring of abc from the index 1 and so on.

Programming Interview Questions 7: Binary Search Tree Check

Given a binary tree, check whether it's a binary search tree or not

```
def isBST(tree, minVal=NEG_INFINITY, maxVal=INFINITY):
    if tree is None:
        return True

    if not minVal <= tree.val <= maxVal:
        return False

    return isBST(tree.left, minVal, tree.val) and \
           isBST(tree.right, tree.val, maxVal)
```

There's an equally good alternative solution. If a tree is a binary search tree, then traversing the tree inorder should lead to sorted order of the values in the tree. So, we can perform an inorder traversal and check whether the node values are sorted or not. Here is the code:

```
def isBST2(tree, lastNode=[NEG_INFINITY]):
    if tree is None:
        return True

    if not isBST2(tree.left, lastNode):
        return False

    if tree.val < lastNode[0]:
        return False

    lastNode[0]=tree.val

    return isBST2(tree.right, lastNode)
```

Programming Interview Questions 8: Transform Word

Given a source word, target word and an English dictionary, transform the source word to target by changing/adding/removing 1 character at a time, while all intermediate words being valid English words. Return the transformation chain which has the smallest number of intermediate words.

We should perform preprocessing on the given dictionary. Let all the words in the dictionary be the nodes of a graph, and there is an undirected edge between two nodes if one word could be converted to another by a single transformation: changing a character, removing a character, or deleting a character. For example there will be an edge between “bat” and “cat” since we can change a character, also between “cat” and “at” by removing a character, and between “bat” and “bart” by adding a character. We can preprocess the whole dictionary and create this huge graph. Assuming that our program will be asked to transform different words to one another many times, the cost of precomputation will be negligible. Here is the code to create the graph from the dictionary, the graph is a hashtable where the key is a word and the value is the list of valid transformations of that word:

```
def constructGraph(dictionary):
    graph=collections.defaultdict(list)
    letters=string.lowercase
    for word in dictionary:
        for i in range(len(word)):
            #remove 1 character
            remove=word[:i]+word[i+1:]
            if remove in dictionary:
                graph[word].append(remove)
            #change 1 character
            for char in letters:
                change=word[:i]+char+word[i+1:]
                if change in dictionary and change!=word:
                    graph[word].append(change)
        #add 1 character
        for i in range(len(word)+1):
            for char in letters:
                add=word[:i]+char+word[i:]
```

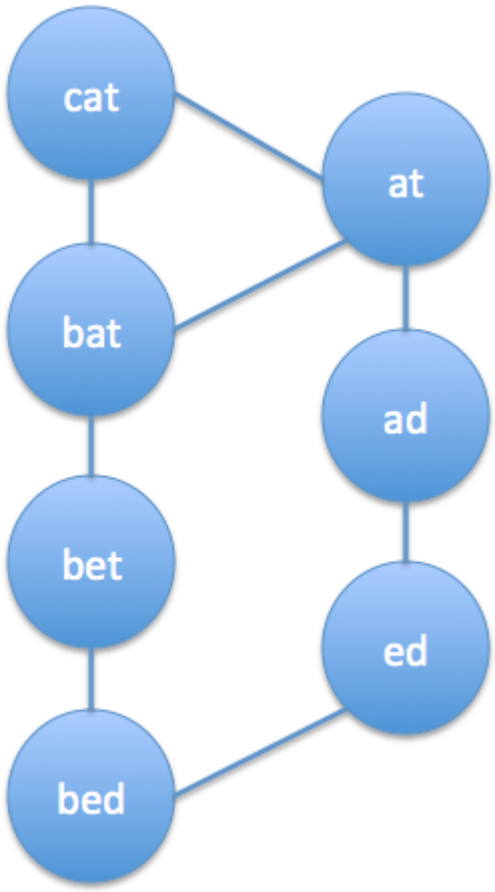
```
                if add in dictionary:
                    graph[word].append(add)

    return graph
```

Now we have the graph where each edge corresponds to a valid transformation between words. So, given two words we can initiate a breadth first search from the start node and once we reach the goal node we will have the shortest chain of transformations between two words (note that either start or goal or both words may not exist in the dictionary at all). Breadth first search gives the shortest path between a start node and a goal node in an unweighted graph, given that the start node is the root of the search. We didn't perform depth first search because it won't necessarily give us the shortest path and we may waste a lot of time trying to explore a dead end because the graph contains many nodes. Here is the code, it takes the graph generated above as an input together with start and goal words:

```
def transformWord(graph, start, goal):
    paths=collections.deque([ [start] ])
    extended=set()
    #Breadth First Search
    while len(paths)!=0:
        currentPath=paths.popleft()
        currentWord=currentPath[-1]
        if currentWord==goal:
            return currentPath
        elif currentWord in extended:
            #already extended this word
            continue

        extended.add(currentWord)
        transforms=graph[currentWord]
        for word in transforms:
            if word not in currentPath:
                #avoid loops
                paths.append(currentPath[:]+[word])

    #no transformation
    return []
```

The complexity of the algorithm depends on how far apart two words are in the graph. In the worst case, we may have to traverse all the graph to find a transformation, or such a transformation may not exist at all, meaning the graph has more than 1 connected components. Or either the source or the goal or both may not exist in the dictionary.

Here is an example, let's say we have the following words in our dictionary: cat, bat, bet, bed, at, ad, ed. The transformation graph is the following:



Let's say we want to find the shortest transformation between the words cat and bed. Here is the demonstration of how it works:

```
>>> dictionary=['cat', 'bat', 'bet', 'bed',
...             'at', 'ad', 'ed']
>>> graph=constructGraph(dictionary)
>>> transformWord(graph, 'cat', 'bed')
['cat', 'bat', 'bet', 'bed']
>>>
>>> dictionary.remove('bet')
>>> graph=constructGraph(dictionary)
>>> transformWord(graph, 'cat', 'bed')
['cat', 'at', 'ad', 'ed', 'bed']
>>>
>>> dictionary.remove('ad')
>>> graph=constructGraph(dictionary)
>>> transformWord(graph, 'cat', 'bed')
```

There are two paths. cat->bat->bet->bed is shorter than cat->at->ad->ed->bed. If we delete the node 'bet' than the first path becomes invalid and the shortest path is now the second one. If we also delete 'ad' then there is no path left.

This question demonstrates the power of graphs and formulating a problem as a graph can reduce a difficult task to a classic traversal algorithm.

Programming Interview Questions 9: Convert Array

Given an array:

$$[a_1, a_2, \ldots, a_N, b_1, b_2, \ldots, b_N, c_1, c_2, \ldots, c_N]$$

convert it to:

$$[a_1, b_1, c_1, a_2, b_2, c_2, \ldots, a_N, b_N, c_N]$$

The element at the ith position in the final array is at position (i%3)*N + i/3 in the original array. So, the code is simply:

```
def getIndex(currentIndex, N):
    return (currentIndex%3)*N + (currentIndex/3)

def convertArray_extraSpace(arr):
    N=len(arr)/3
    return [arr[getIndex(i, N)] for i in range(len(arr))]
```

The getIndex function takes an index from the final array, and returns the index of the element in the original array that should appear at that position. However, we aren't

allowed use extra space, we should instead modify the array in-place. We could use a similar approach though, at each iteration we can put the ith element to its final location using the getIndex function above and swap elements. The algorithm works as follows, at each iteration (currentIndex) we get the index of the item that should appear at that location (swapIndex) by calling the getIndex function. The element at swapIndex is the final element to appear at currentIndex. So we swap the elements at currentIndex and swapIndex, if swapIndex>=currentIndex. But if swapIndex<currentIndex then it means that the element at swapIndex was replaced with another element at previous iterations. Now it's somewhere else and we should keep looking for that element. We again call getIndex with swapIndex as new input to find the element it was replaced with. If the new swapIndex>=currentIndex, we swap the elements as before. Otherwise, we repeat this procedure until swapIndex>=currentIndex, which is we find the final element that's supposed to appear at currentIndex.

The algorithm is pretty simple and in-place without using extra space. Let's work through an example to clarify, here is the program flow for an array of size 15. Swap index is calculated multiple times for some elements until swapIndex>=currentIndex as explained above.

```
currentIdx=0, swapIdx=0, swapIdx>=currentIdx, swap arr[0] arr[0]
currentIdx=1, swapIdx=5, swapIdx>=currentIdx, swap arr[1] arr[5]
currentIdx=2, swapIdx=10, swapIdx>=currentIdx, swap arr[2] arr[10]
currentIdx=3, swapIdx=1, swapIdx<currentIdx, no swap
  swapIdx=1, newSwapIdx=5, newSwapIdx>=currentIdx, swap arr[3]
arr[5]
currentIdx=4, swapIdx=6, swapIdx>=currentIdx, swap arr[4] arr[6]
currentIdx=5, swapIdx=11, swapIdx>=currentIdx, swap arr[5] arr[11]
currentIdx=6, swapIdx=2, swapIdx<currentIdx, no swap
  swapIdx=2, newSwapIdx=10, newSwapIdx>=currentIdx, swap arr[6]
arr[10]
currentIdx=7, swapIdx=7, swapIdx>=currentIdx, swap arr[7] arr[7]
currentIdx=8, swapIdx=12, swapIdx>=currentIdx, swap arr[8] arr[12]
currentIdx=9, swapIdx=3, swapIdx<currentIdx, no swap
  swapIdx=3, newSwapIdx=1, newSwapIdx<currentIdx, no swap
  swapIdx=1, newSwapIdx=5, newSwapIdx<currentIdx, no swap
  swapIdx=5, newSwapIdx=11, newSwapIdx>=currentIdx, swap arr[9]
arr[11]
currentIdx=10, swapIdx=8, swapIdx<currentIdx, no swap
```

```
  swapIdx=8, newSwapIdx=12, newSwapIdx>=currentIdx, swap arr[10]
arr[12]
currentIdx=11, swapIdx=13, swapIdx>=currentIdx, swap arr[11]
arr[13]
currentIdx=12, swapIdx=4, swapIdx<currentIdx, no swap
  swapIdx=4, newSwapIdx=6, newSwapIdx<currentIdx, no swap
  swapIdx=6, newSwapIdx=2, newSwapIdx<currentIdx, no swap
  swapIdx=2, newSwapIdx=10, newSwapIdx<currentIdx, no swap
  swapIdx=10, newSwapIdx=8, newSwapIdx<currentIdx, no swap
  swapIdx=8, newSwapIdx=12, newSwapIdx>=currentIdx, swap arr[12]
arr[12]
currentIdx=13, swapIdx=9, swapIdx<currentIdx, no swap
  swapIdx=9, newSwapIdx=3, newSwapIdx<currentIdx, no swap
  swapIdx=3, newSwapIdx=1, newSwapIdx<currentIdx, no swap
  swapIdx=1, newSwapIdx=5, newSwapIdx<currentIdx, no swap
  swapIdx=5, newSwapIdx=11, newSwapIdx<currentIdx, no swap
  swapIdx=13, newSwapIdx=13, newSwapIdx>=currentIdx, swap arr[13]
arr[13]
currentIdx=14, swapIdx=14, currentIdx>=swapIdx, swap arr[14]
arr[14]
```

Programming Interview Questions 10: Kth Largest Element in Array

Given an array of integers find the kth element in the sorted order (not the kth distinct element). So, if the array is [3, 1, 2, 1, 4] and k is 3 then the result is 2, because it's the 3rd element in sorted order (but the 3rd distinct element is 3).

The first approach that comes to mind is sorting the array and returning the kth element. The complexity is NlogN where N is size of the array and it's clearly not optimal. Because this solution does more work than needed, it finds the absolute ordering of all elements but we're only looking for the kth largest element. We would ideally prefer a linear time solution.

Programming Interview Questions 11: All Permutations of String

Generate all permutations of a given string.

Let the string be ‘LSE’, and we have length 2 permutations ‘SE’ and ‘ES’. How do we incorporate the letter L into these permutations? We just insert it into every possible location in both strings: beginning, middle, and the end. So for ‘SE’ the result is: ‘LSE’, ‘SLE’, ‘SEL’. And for the string ‘ES’ the results is: ‘LES’, ‘ELS’, ‘ESL’. We inserted the character L to every possible location in all the strings. This is it!. We will just use a recursive algorithm and we’re done

We remove the first character and recurse to get all permutations of length N-1, then we insert that first character into N-1 length strings and obtain all permutations of length N . The complexity is O(N!) because there are N! possible permutations of a string with length N, so it’s optimal.

```
def permutations(word):
    if len(word)<=1:
        return [word]

    #get all permutations of length N-1
    perms=permutations(word[1:])
    char=word[0]
    result=[]
    #iterate over all permutations of length N-1
    for perm in perms:
        #insert the character into every possible location
        for i in range(len(perm)+1):
            result.append(perm[:i] + char + perm[i:])
    return result
```

Programming Interview Questions 12: Reverse Words in a String

Given an input string, reverse all the words. To clarify, input: “Interviews are awesome!” output: “awesome! are Interviews”. Consider all consecutive non-whitespace characters as individual words. If there are multiple spaces between words reduce them to a single white space. Also remove all leading and trailing whitespaces. So, the output for ” CS degree”, “CS degree”, “CS degree “, or ” CS degree ” are all the same: “degree CS”.

Reverse all the characters in the string, then reverse the letters of each individual word. This can be done in-place using C or C++.

In C/C++ we would first reverse the entire string and loop over it with two pointers, read and write. We’ll overwrite the string in-place. The resulting string may be shorter than the original one, because we have to remove multiple consecutive spaces as well as leading and trailing ones, that’s why we need 2 pointers. But note that write pointer can never pass read pointer so there won’t be any conflicts. Here is the C code:

```
void reverseWords(char *text)
{
    int length=strlen(text);
    reverseString(text, 0, length-1, 0);
    int read=0, write=0;
    while (read<length)
    {
        if (text[read]!=' ')
        {
            int wordStart=read;
            for ( ;read<length && text[read]!=' '; read++);
            reverseString(text, wordStart, read-1, write);
            write+=read-wordStart;
            text[write++]=' ';
        }
        read++;
    }
    text[write-1]='\0';
}

void reverseString(char *text, int start, int end, int destination)
{
    // reverse the string and copy it to destination
    int length=end-start+1;
    int i;
```

```
    memcpy(&text[destination], &text[start], length*sizeof(char));
    for (i=0; i<length/2; i++)
    {
        swap(&text[destination+i], &text[destination+length-1-i]);
    }
}
```

Programming Interview Questions 13: Median of Integer Stream

Given a stream of unsorted integers, find the median element in sorted order at any given time.

So, we will be receiving a continuous stream of numbers in some random order and we don't know the stream length in advance. Write a function that finds the median of the already received numbers efficiently at any time. We will be asked to find the median multiple times. Just to recall, median is the middle element in an odd length sorted array, and in the even case it's the average of the middle elements.

Following approaches:

1. We could insert the integers to an unsorted array. Inserting will take O(1) time while finding the median will take O(N) time using the Median of Medians algorithm. Since the algorithm does not find the median efficiently, it cannot be considered.
2. We could use a sorted array. Finding the position where we want to insert will take us
   O (logN) time, but after finding the postion, we have to shift the elements to the right. This will take O(N) complexity. So inserting will take O(logN) time , the overall insertion complexity is O(N) due to shifting. we can just return the middle element if the array length is odd, or the average of middle elements if the length is even. This can be done in O(1) time, which is exactly what we're looking for. But finding the median is still extremely efficient, constant time. However, linear time insertion is pretty inefficient and we would prefer a better performance.

3. If we use unsorted linked list, then we can insert into either as head or tail which is O(N) but we suffer the same problem of unsorted array where finding the median is O(N).
4. Insertion into a sorted linked list is O(N) because we cannot perform binary search in a linked list even if it is sorted. So, using a sorted linked list doesn't worth the effort, insertion is O(N) and finding median is O(1), same as the sorted array.
5. If we use a binary tree with additional information such as number of nodes in the tree, then we can find the median in O(logN) time, taking the appropriate branch in the tree based on number of children on the left and right of the current node. However, the insertion complexity is O(N) because a standard binary search tree can degenerate into a linked list if we happen to receive the numbers in sorted order.
6. Balanced binary search trees seem to be the most optimal solution, insertion is O(logN) and find median is O(1).

Solution

We will use a max heap and a min heap with 2 requirements.

1. The first requirement is that the max heap contains the smallest half of the numbers and min heap contains the largest half. So, the numbers in max-heap are always less than or equal to the numbers in min-heap.
2. The second requirement is that the number of elements in max-heap is either equal or 1 more than the number of elements in the min heap. So, if we received 2N elements (even) up to now, max-heap and min-heap will both contain N elements. Otherwise, if we have received 2N+1 elements (odd), max-heap will contain N+1 and min-heap N.

The heaps are constructed considering the two requirements above. Then once we're asked for the median, if the total number of received elements is odd, the median is the root of the max-heap. If it's even, then the median is the average of the roots of the max-heap and min-heap.

1. When the number of elements in even, then the element is inserted into the max heap. If the new element is less than all the elements in the min heap, then we are done. If the new element is greater than the root of the

min heap, then this element will be the root of the max heap as well. Now to follow the first requirement, we need to swap the roots of max heap and min heap such that the bottom half belongs to max heap and the top half belongs to min heap.

2. If the number of elements in odd, then the new element is inserted into the max heap. We then pop the root element from the max heap and push it to the min heap so the first requirement is met.

Insertion complexity is O(logN) and finding the median is O(1).

```
class streamMedian:
    def __init__(self):
        self.minHeap, self.maxHeap = [], []
        self.N=0

    def insert(self, num):
        if self.N%2==0:
            heapq.heappush(self.maxHeap, -1*num)
            self.N+=1
            if len(self.minHeap)==0:
                return
            if -1*self.maxHeap[0]>self.minHeap[0]:
                toMin=-1*heapq.heappop(self.maxHeap)
                toMax=heapq.heappop(self.minHeap)
                heapq.heappush(self.maxHeap, -1*toMax)
                heapq.heappush(self.minHeap, toMin)
        else:
            toMin=-1*heapq.heappushpop(self.maxHeap, -1*num)
            heapq.heappush(self.minHeap, toMin)
            self.N+=1

    def getMedian(self):
        if self.N%2==0:
            return (-1*self.maxHeap[0]+self.minHeap[0])/2.0
        else:
            return -1*self.maxHeap[0]
```

Programming Interview Questions 14: Check Balanced Parentheses

Given a string of opening and closing parentheses, check whether it's balanced. We have 3 types of parentheses: round brackets: (), square brackets: [], and curly brackets: {}. Assume that the string doesn't contain any other character than these, no spaces words or numbers. Just to remind, balanced parentheses require every opening parenthesis to be closed in the reverse order opened. For example '([])' is balanced but '([)]' is not.

We scan the string from left to right, and every time we see an opening parenthesis we push it to a stack, because we want the last opening parenthesis to be closed first. Then, when we see a closing parenthesis we check whether the last opened one is the corresponding closing match, by popping an element from the stack. If it's a valid match, then we proceed forward, if not return false. Or if the stack is empty we also return false, because there's no opening parenthesis associated with this closing one. In the end, we also check whether the stack is empty. If so, we return true, otherwise return false because there were some opened parenthesis that were not closed.

```
opening=set('([{')
    match=set([ ('(',')'), ('[',']'), ('{','}') ])
    stack=[]
    for char in expr:
        if char in opening:
            stack.append(char)
        else:
            if len(stack)==0:
                return False
            lastOpen=stack.pop()
            if (lastOpen, char) not in match:
                return False
    return len(stack)==0
```

Programming Interview Questions 15: First Non Repeated Character in String

Find the first non-repeated (unique) character in a given string.

This question demonstrates efficient use of hashtable. We scan the string from left to right counting the number occurrences of each character in a hashtable. Then we perform a second pass and check the counts of every character. Whenever we hit a count of 1 we return that character, that's the first unique letter.

```
def firstUnique(text):
    counts=collections.defaultdict(int)
    for letter in text:
        counts[letter]+=1
    for letter in text:
        if counts[letter]==1:
            return letter
```

As you can see it's pretty straightforward once we use a hashtable. It's an optimal solution, the complexity is O(N). Hashtable is generally the key data structure to achieve optimal linear time solutions in questions that involve counting

Programming Interview Questions 16: Anagram Strings

Given two strings, check if they're anagrams or not. Two strings are anagrams if they are written using the same exact letters, ignoring space, punctuation and capitalization. Each letter should have the same count in both strings. For example, 'Eleven plus two' and 'Twelve plus one' are meaningful anagrams of each other.

First we should extract only the letters from both strings and convert to lowercase, excluding punctuation and whitespaces. Then we can compare these to check whether two strings are anagrams of each other. From now on when I refer to a string, I assume this transformation is performed and it only contains lowercase letters in original order. If two strings contain every letter same number of times, then they are anagrams.

One way to perform this check is to sort both strings and check whether they're the same or not. The complexity is O(NlogN) where N is the number of characters in the string.
Sorting approach is elegant but not optimal. We would prefer a linear time solution. Since the problem involves counting, hashtable would be a suitable data structure. We can store the counts of each character in string1 in a hashtable. Then we scan string2 from left to right decreasing the count of each letter. Once the count becomes negative (string2 contains more of that character) or if the letter doesn't exist in the hashtable (string1 doesn't contain that character), then the strings are not anagrams. Finally we check whether all the counts in the hashtable are 0, otherwise string1 contains extra characters. Or we can check the lengths of the strings in the beginning and avoid this count check. This also allows early termination of the program if the strings are of different lengths, because they can't be anagrams. The code is the following:

```
def isAnagram2(str1, str2):
    str1, str2 = getLetters(str1), getLetters(str2)
    if len(str1)!=len(str2):
        return False
    counts=collections.defaultdict(int)
    for letter in str1:
        counts[letter]+=1
    for letter in str2:
        counts[letter]-=1
        if counts[letter]<0:
            return False
    return True
```

Programming Interview Questions 17: Search Unknown Length Array

Given a sorted array of unknown length and a number to search for, return the index of the number in the array. Accessing an element out of bounds throws exception. If the

number occurs multiple times, return the index of any occurrence. If it isn't present, return -1.

Standard binary search wouldn't work because we don't know the size of the array to provide an upper limit index. So, we perform one-sided binary search for both the size of the array and the element itself simultaneously. Let's say we're searching for the value k. We check array indexes 0, 1, 2, 4, 8, 16, ..., 2^N in a loop until either we get an exception or we see an element larger than k. If the value is less than k we continue, or if we luckily find the actual value k then we return the index

If at index 2^m we see an element larger than k, it means the value k (if it exists) must be between indexes 2^(m-1)+1 and 2^m-1 (inclusive), since the array is sorted. The same is true if we get an exception, because we know that the number at index 2^(m-1) is less than k, and we didn't get an exception accessing that index. Getting an exception at index 2^m means the size of the array is somewhere between 2^(m-1) and 2^m-1. In both cases we break out of the loop and start another modified binary search, this time between indexes 2^(m-1)+1 and 2^m-1. If we previously got exception at index 2^m, we may get more exceptions during this binary search so we should handle this case by assigning the new high index to that location

Programming Interview Questions 18: Find Even Occurring Element

Given an integer array, one element occurs even number of times and all others have odd occurrences. Find the element with even occurrences. We can use a hashtable as we always do with problems that involve counting. Scan the array and count the occurrences of each number. Then perform a second pass from the hashtable and return the element with even count. Here's the code:

```
def getEven1(arr):
    counts=collections.defaultdict(int)
    for num in arr:
        counts[num]+=1
    for num, count in counts.items():
        if count%2==0:
            return num
```

This can also be solved using XOR. First we get all the unique numbers in the array using a set in O(N) time. Then we XOR the original array and the unique numbers all together. Result of XOR is the even occurring element. Because every odd

occurring element in the array will be XORed with itself odd number of times, therefore producing a 0. And the only even occurring element will be XORed with itself even number of times, which is the number itself. The order of XOR is not important. The conclusion is that if we XOR a number with itself odd number of times we get 0, otherwise if we XOR even number of times then we get the number itself. And with multiple numbers, the order of XOR is not important, just how many times we XOR a number with itself is significant.

Add all the numbers to the set, since sets contain only unique elements by definition, we obtain the unique numbers

Programming Interview Questions 19: Find Next Palindrome Number

Given a number, find the next smallest palindrome larger than the number. For example if the number is 125, next smallest palindrome is 131.

There are two cases, whether the number of digits in the number is odd or even. We'll start with analyzing the odd case

Let's say the number is ABCDE, the smallest possible palindrome we can obtain from this number is ABCBA, which is formed by mirroring the number around its center from left to right (copying the left half onto the right in reverse order). This number is always a palindrome, but it may not be greater than the original number. If it is then we simply return the number,
we're looking for the smallest palindrome larger than the number. The digits on the right half increase the value of the number much less, but then to keep the number palindrome we'll have to increment the digits on the left half as well, which will again result in a large increase. So, we increment the digit just in the middle by 1. Copy the left half into the right in reverse order, so the number is a palindrome.

Let's say the given number is 250, we first take the mirror image around its center, resulting in 252. 252 is greater than 250 so this is the first palindrome greater than the given number, we're done.

Now let's say the number is 123, now mirroring the number results in 121, which is less than the original number. So we increment it's middle digit, resulting in 131.

If the middle digit is 9, then the solution is we first round up the number and then applies the procedure to it. For example if the number is 397, mirroring results in 393 which is less. So we round it up to 400 and solve the problem as if we got 400 in the first place. We take the mirror image, which is 404 and this is the result.

Now let's analyze the case where the given number has even number of digits. Get the middle two digits of the number. Select the first middle digit and copy the left half into the right in reverse order.

Assume the given number is 4512, we mirror the number around its center, resulting in 4554. This is greater than the given number so we're done.

Now let the number be 1234, mirroring results in 1221 which is less than the original number. So we increment the middle two digits, resulting in 1331 which is the result.

if the given number is 1997 mirroring would give 1991, which is less. So we round it up to 2000 and solve as if it were the original number. We mirror it, resulting in 2002 and this is the result.

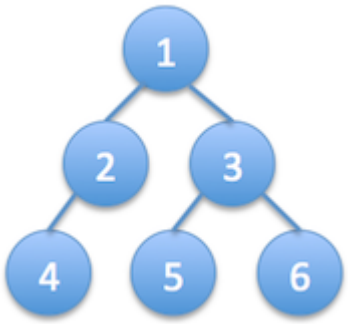
```
def nextPalindrome(num):
    length=len(str(num))
    oddDigits=(length%2!=0)
    leftHalf=getLeftHalf(num)
    middle=getMiddle(num)
    if oddDigits:
        increment=pow(10, length/2)
        newNum=int(leftHalf+middle+leftHalf[::-1])
    else:
        increment=int(1.1*pow(10, length/2))
        newNum=int(leftHalf+leftHalf[::-1])
    if newNum>num:
        return newNum
    if middle!='9':
        return newNum+increment
    else:
        return nextPalindrome(roundUp(num))
```

```
def getLeftHalf(num):
    return str(num)[:len(str(num))/2]

def getMiddle(num):
    return str(num)[(len(str(num))-1)/2]

def roundUp(num):
    length=len(str(num))
    increment=pow(10,((length/2)+1))
    return ((num/increment)+1)*increment
```

The complexity of this algorithm is O(logN), because the complexity of mirroring is proportional to the number of digits in the number, which is the ceiling of logN.

Programming Interview Questions 20: Tree Level Order Print

Given a binary tree of integers, print it in level order. The output will contain space between the numbers in the same level, and new line between different levels. For example, if the tree is:



The output should be:
1
2 3
4 5 6

It won't be practical to solve this problem using recursion, because recursion is similar to depth first search, but what we need here is breadth first search. So we

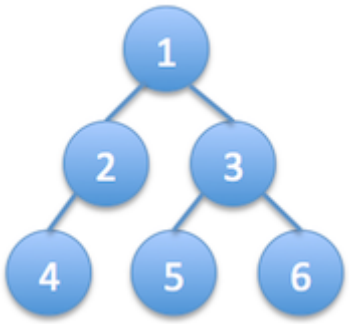
will use a queue as we did previously in breadth first search. First, we'll push the root node into the queue. Then we start a while loop with the condition queue not being empty. Then, at each iteration we pop a node from the beginning of the queue and push its children to the end of the queue. Once we pop a node we print its value and space.

To print the new line in correct place we should count the number of nodes at each level. We will have 2 counts, namely current level count and next level count. Current level count indicates how many nodes should be printed at this level before printing a new line. We decrement it every time we pop an element from the queue and print it. Once the current level count reaches zero we print a new line. Next level count contains the number of nodes in the next level, which will become the current level count after printing a new line. We count the number of nodes in the next level by counting the number of children of the nodes in the current level. Understanding the code is easier than its explanation:

```
currentCount, nextCount = 1, 0
    while len(nodes)!=0:
        currentNode=nodes.popleft()
        currentCount-=1
        print currentNode.val,
        if currentNode.left:
            nodes.append(currentNode.left)
            nextCount+=1
        if currentNode.right:
            nodes.append(currentNode.right)
            nextCount+=1
        if currentCount==0:
            #finished printing current level
            print '\n',
        currentCount = nextCount
```

Programming Interview Questions 21: Tree Reverse Level Order Print

We again print the tree in level order, but now starting from bottom level to the root. Using the same tree as before:



The output should be:
4 5 6
2 3
1

The solution of this problem is similar to level order print. We start a breadth first search (BFS) from the root of the tree and push each node to a queue. We don't print any node at this point because we want to output bottom up, but BFS progresses from top to bottom. Printing will be take place a separate loop after completing breadth first search. We also count the number of nodes in each level and push them to a stack, in order to print the new lines in correct places. So after completion of BFS we have the following two data structures:

Queue of nodes: [1, 2, 3, 4, 5, 6]. This queue is constructed by BFS from top to bottom and left to right.
Stack of node counts at each level: [3, 2, 1]. Note that since this is a stack the first element is the node count at the deepest level, and the last count is always 1 which corresponds to the root of the tree.

After constructing these data structures, the nodes we want to print as the first line of output are at the end of the queue. And the number of nodes to print is at the top of the stack. So we start a loop where at each iteration we pop an element from the stack and print that many number of nodes from the end of the queue. Using the example tree above, the first line of the output contains 3 nodes, the first element in the stack. And the nodes to print are the 3 nodes at the end of the queue, which is [4, 5, 6]. The second line of the output contains 2 nodes, the second value in the stack. These are the 2 nodes in the queue that are just before the first line nodes, namely [2, 3]. Finally, the number of nodes in the last line is at the end of the stack, which is 1. And the node to print is at the beginning of the queue, the value 1.

```
while i<len(nodes):
        currentNode=nodes[i]
```

```
        currentCount-=1
        if currentNode.left:
            nodes.append(currentNode.left)
            nextCount+=1
        if currentNode.right:
            nodes.append(currentNode.right)
            nextCount+=1
        if currentCount==0:
            #finished this level
            if nextCount==0:
                #no more nodes at next level
                break
            #continue with next level
            levelCount.appendleft(nextCount)
            currentCount, nextCount = nextCount, currentCount
        i+=1
    printIndex=len(nodes)
    for count in levelCount:
        output=nodes[printIndex-count:printIndex]
        print ' '.join(map(str, output)), '\n',
        printIndex-=count
```

Programming Interview Questions 22: Find Odd Occurring Element

Given an integer array, one element occurs odd number of times and all others have even occurrences. Find the element with odd occurrences.

One approach is again to build a hashtable of element occurrence counts and return the element with odd count. Both time and space complexity of this solution is O(N).

But we can do much better by using the XOR trick described in that post. It's the following: if we XOR a number with itself odd number of times the result is 0, otherwise if we XOR even number of times the result is the number itself. So, if we XOR all the elements in the array, the result is the odd occurring element itself. Because all even occurring elements will be XORed with themselves odd

number of times, producing 0. And the only odd occurring element will be XORed with itself even number of times, producing its own value.

Programming Interview Questions 23: Find Word Positions in Text

Given a text file and a word, find the positions that the word occurs in the file. We'll be asked to find the positions of many words in the same file.

Better Solution

Using hashtable as the inverted index is pretty efficient. But we can do better by using a more space efficient data structure for text, namely a trie, also known as prefix tree. Trie is a tree which is very efficient in text storage and search. It saves space by letting the words that have the same prefixes to share data. And most languages contain many words that share the same prefix.

Let's demonstrate this with an example using the words subtree, subset, and submit. In a hashtable index, each of these words will be stored separately as individual keys. So, it will take 7+6+6=19 characters of space in the index. But all these words share the same prefix 'sub'. We can take advantage of this fact by using a trie and letting them share that prefix in the index. In a trie index these words will take 7+3+3=13 characters of space, cutting the size of the index around 30%. We can have even more gain with the words that share longer prefixes. For example author, authority, and authorize. Hashtable index uses 6+9+9=24 characters but trie uses only 6+3+2=11, leading to 55% compression. Considering the extremely big web indexes of search engines, reducing the index size even by 10% is a big gain, which results in saving terabytes of space and reduces the number of machines by 1/10. Which is huge given that thousands of machines are used at Google, Microsoft, and Yahoo.

A trie is simply a tree where each node stores a character as key, and the value in our case will be the occurrence positions of the word associated with the node. The word associated with a node is concatenation of the characters from root of the tree to the node
Maximum number of children of a node is the number of different characters that appear in the text, which is 36 if we only consider lowercase alphanumeric characters.

Here is the structure of a node:

```
class Node:
    def __init__(self, letter):
        self.letter=letter
        self.isTerminal=False
        self.children={}
        self.positions=[]
```

The trie data structure is composed of these nodes, and it'll include the following 3 functions: getItem, contains, and output.
The getItem function both returns the occurrence positions of a given word and it's used for insertion, contains function checks whether a word is in the trie or not, and output prints the trie in a nice formatted manner.
We insert a new occurrence position to a word by first getting its existing position list using the getItem function, and then appending the new position to the end of the list.

Trying to access an element that's not in the trie automatically creates that element, similar to collections.defaultdict.
Code of getItem and contains functions are very similar. Both navigate through the tree until they reach the node that corresponds to the given word. Output prints the words in the trie in sorted order and indented in a way that prefix relations can be visually identified.

New createIndex and queryIndex functions are very similar to hashtable ones:

```
def createIndex2(text):
    trie=Trie()
    words=getWords(text)
    for pos, word in enumerate(words):
        trie[word].append(pos)
    return trie

def queryIndex2(index, word):
    if word in index:
        return index[word]
    else:
        return []
```

Programming Interview Questions 24: Find Next Higher Number With Same Digits

Given a number, find the next higher number using only the digits in the given number. For example if the given number is 1234, next higher number with same digits is 1243.

Let's visualize a better solution using an example, the given number is 12543 and the resulting next higher number should be 13245. We scan the digits of the given number starting from the tenths digit (which is 4 in our case) going towards left. At each iteration we check the right digit of the current digit we're at, and if the value of right is greater than current we stop, otherwise we continue to left. So we start with current digit 4, right digit is 3, and 4>=3 so we continue. Now current digit is 5, right digit is 4, and 5>= 4, continue. Now current is 2, right is 5, but it's not 2>=5, so we stop. The digit 2 is our pivot digit. From the digits to the right of 2, we find the smallest digit higher than 2, which is 3. This part is important, we should find the smallest higher digit for the resulting number to be precisely the next higher than original number. We swap this digit and the pivot digit, so the number becomes 13542. Pivot digit is now 3. We sort all the digits to the right of the pivot digit in increasing order, resulting in 13245. This is it, here's the code:

```
def nextHigher(num):
    strNum=str(num)
    length=len(strNum)
    for i in range(length-2, -1, -1):
        current=strNum[i]
        right=strNum[i+1]
        if current<right:
            temp=sorted(strNum[i:])
            next=temp[temp.index(current)+1]
            temp.remove(next)
            temp=''.join(temp)
            return int(strNum[:i]+next+temp)
    return num
```

Note that if the digits of the given number is monotonically increasing from right to left, like 43221 then we won't perform any operations, which is what we want because this is the highest number obtainable from these digits. There's no higher number, so we

return the given number itself. The same case occurs when the number has only a single digit, like 7. We can't form a different number since there's only a single digit.

The complexity of this algorithm also depends on the number of digits, and the sorting part dominates. A given number N has logN+1 digits and in the worst case we'll have to sort logN digits. Which happens when all digits are increasing from right to left except the leftmost digit, for example 1987. For sorting we don't have to use comparison based algorithms such as quicksort, mergesort, or heapsort which are O(KlogK), where K is the number of elements to sort. Since we know that digits are always between 0 and 9, we can use counting sort, radix sort, or bucket sort which can work in linear time O(K). So the overall complexity of sorting logN digits will stay linear resulting in overall complexity O(logN). Which is optimal since we have to check each digit at least once.

Programming Interview Questions 25: Remove Duplicate Characters in String

Remove duplicate characters in a given string keeping only the first occurrences. For example, if the input is 'tree traversal' the output will be 'tre avsl'.

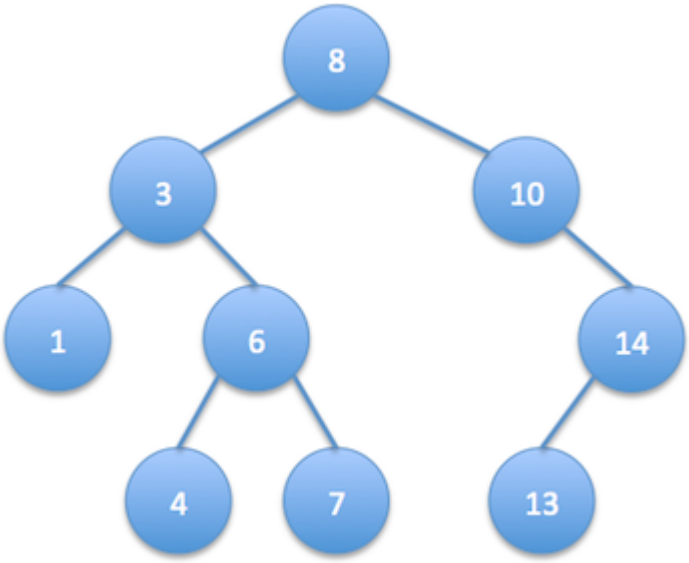
We need a data structure to keep track of the characters we have seen so far, which can perform efficient find operation. If the input is guaranteed to be in standard ASCII form, we can just create a boolean array of size 128 and perform lookups by accessing the index of the character's ASCII value in constant time. But if the string is Unicode then we would need a much larger array of size more than 100K, which will be a waste since most of it would generally be unused.

Set data structure perfectly suits our purpose. It stores keys and provides constant time search for key existence.

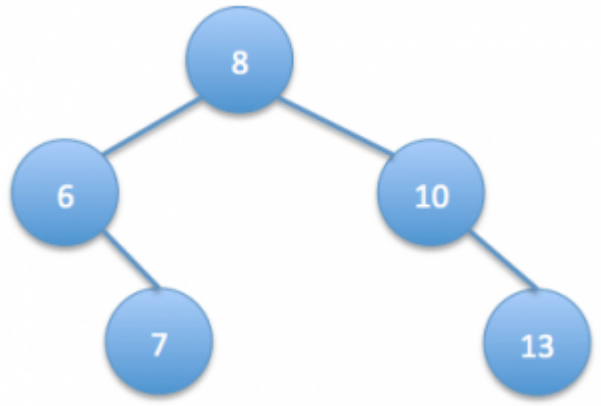
It stores keys and provides constant time search for key existence. So, we'll loop over the characters of the string, and at each iteration we'll check whether we have seen the current character before by searching the set. If it's in the set then it means we've seen it before, so we ignore it. Otherwise, we include it in the result and add it to the set to keep track for future reference.

Programming Interview Questions 26: Trim Binary Search Tree

Given the root of a binary search tree and 2 numbers min and max, trim the tree such that all the numbers in the new tree are between min and max (inclusive). The resulting tree should still be a valid binary search tree. So, if we get this tree as input:



and we're given min value as 5 and max value as 13, then the resulting binary search tree should be:



If current node's value is between min and max (min<=node<=max) then there's no action need to be taken, so we return the reference to the node itself. If current node's value is less than min, then we return the reference to its right subtree, and discard the left subtree. Because if a node's value is less than min, then its left children are definitely less than min since this is a binary search tree. But its right children may or may not be less than min we can't be sure, so we return the reference to it. Since we're performing bottom-up post-order traversal, its right subtree is already a trimmed valid binary

search tree (possibly NULL), and left subtree is definitely NULL because those nodes were surely less than min and they were eliminated during the post-order traversal. Remember that in post-order traversal we first process all the children of a node, and then finally the node itself.

```
def trimBST(tree, minVal, maxVal):
    if not tree:
        return
    tree.left=trimBST(tree.left, minVal, maxVal)
    tree.right=trimBST(tree.right, minVal, maxVal)
    if minVal<=tree.val<=maxVal:
        return tree
    if tree.val<minVal:
        return tree.right
    if tree.val>maxVal:
        return tree.left
```

Programming Interview Questions 27: Squareroot of a Number

Find the squareroot of a given number rounded down to the nearest integer, without using the sqrt function. For example, squareroot of a number between [9, 15] should return 3, and [16, 24] should be 4.

The squareroot of a (non-negative) number N always lies between 0 and N/2.
We know that the result is between 0 and N/2, so we can first try N/4 to see whether its square is less than, greater than, or equal to N. If it's equal then we simply return that value. If it's less, then we continue our search between N/4 and N/2. Otherwise if it's greater, then we search between 0 and N/4. In both cases we reduce the potential range by half and continue, this is the logic of binary search. We're not performing regular binary search though, it's modified. We want to ensure that we stop at a number k, where k^2<=N but (k+1)^2>N. The code will clarify any doubts:

```
def sqrt2(num):
    if num<0:
        raise ValueError
```

```
    if num==1:
        return 1
    low=0
    high=1+(num/2)
    while low+1<high:
        mid=low+(high-low)/2
        square=mid**2
        if square==num:
            return mid
        elif square<num:
            low=mid
        else:
            high=mid
    return low
```

One difference from regular binary search is the condition of the while loop, it's low+1<high instead of low<high. Also we have low=mid instead of low=mid+1, and high=mid instead of high=mid-1. These are the modifications we make to standard binary search. The complexity is still the same though, it's logarithmic O(logN). Which is much better than the naive linear solution.

Programming Interview Questions 28: Longest Compound Word

Given a sorted list of words, find the longest compound word in the list that is constructed by concatenating the words in the list. For example, if the input list is: ['cat', 'cats', 'catsdogcats', 'catxdogcatsrat', 'dog', 'dogcatsdog', 'hippopotamuses', 'rat', 'ratcat', 'ratcatdog', 'ratcatdogcat']. Then the longest compound word is 'ratcatdogcat' with 12 letters. Note that the longest individual words are 'catxdogcatsrat' and 'hippopotamuses' with 14 letters, but they're not fully constructed by other words. Former one has an extra 'x' letter, and latter is an individual word by itself not a compound word.

We will use the trie data structure also known as the prefix tree.

Trie definition:

```
class Node:
    def __init__(self, letter=None, isTerminal=False):
        self.letter=letter
```

```
        self.children={}
        self.isTerminal=isTerminal
```

The isTerminal field in a Node class means that the word constructed by letters from the root to the current node is a valid word. For example the word cat is formed by 3 nodes, one for each letter. And the last letter ‘t’ has isTerminal value True.

Insert into a Trie

```
def insert(self, word):
    current=self.root
    for letter in word:
        if letter not in current.children:
            current.children[letter]=Node(letter)
        current=current.children[letter]
    current.isTerminal=True
```

The insert function of the trie just adds the given word to the trie.

GetallPrefixes of the trie

The more important getAllPrefixesOfWord function takes a word as input, and returns all the valid prefixes of that word where valid means that prefix word appears in the given input list.

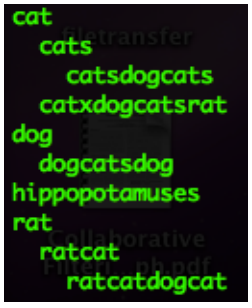
```
def getAllPrefixesOfWord(self, word):
    prefix=''
    prefixes=[]
    current=self.root
    for letter in word:
        if letter not in current.children:
            return prefixes
        current=current.children[letter]
        prefix+=letter
        if current.isTerminal:
            prefixes.append(prefix)
    return prefixes
```

While scanning the list from the beginning, all the prefixes of the current word have already been seen and added to the trie.

So finding all the prefixes of a word can be done easily by following a single path down from the root with nodes being the letters of the word, and returning the prefix words corresponding to nodes with true isTerminal value

The algorithm works as follows, we scan the input list from the beginning and insert each word into the trie. Right after inserting a word, we check whether it has any prefixes. If yes, then it's a candidate for the longest compound word. We append a pair of current word and its suffix (word-prefix) into a queue. The reason is that the current word is a valid construct only if the suffix is also a valid word or a compound word. So we build the trie and queue while scanning the array.

Let's illustrate the process with an example, the first word is cat and we add it to the trie. Since it doesn't have a prefix, we continue. Second word is cats, we add it to the trie and check whether it has a prefix, and yes it does, the word cat. So we append the pair <'cats', 's'> to the queue, which is the current word and the suffix. The third word is catsdogcats, we again insert it to the trie and see that it has 2 prefixes, cat and cats. So we add 2 pairs <'catsdogcats', 'sdogcats'> and <'catsdogcats', 'dogcats'> where the former suffix correspond to the prefix cat and the latter to cats. We continue like this by adding <'catxdogcatsrat', 'xdogcatsrat'> to the queue and so on. And here's the trie formed by adding example the words in the problem definition:



After building the trie and the queue, then we start processing the queue by popping a pair from the beginning. As explained above, the pair contains the original word and the suffix we want to validate. We check whether the suffix is a

valid or compound word. If it's a valid word and the original word is the longest up to now, we store the result. Otherwise we discard the pair. The suffix may be a compound word itself, so we check if the it has any prefixes. If it does, then we apply the above procedure by adding the original word and the new suffix to the queue. If the suffix of the original popped pair is neither a valid word nor has a prefix, we simply discard that pair.

An example will clarify the procedure, let's check the pair <'catsdogcats', 'dogcats'> popped from the queue. Dogcats is not a valid word, so we check whether it has a prefix. Dog is a prefix, and cats is the new suffix. So we add the pair <'catsdogcats', 'cats'> to the queue. Next time we pop this pair we'll see that cats is a valid word and finish processing the word catsdogcats. As you can see, the suffix will get shorter and shorter at each iteration, and at some point the pair will either be discarded or stored as the longest word. And as the pairs are being discarded, the length of the queue will decrease and the algorithm will gradually come to an end. Here's the complete algorithm:

```
def longestWord(words):
    #Add words to the trie, and pairs to the queue
    trie=Trie()
    queue=collections.deque()
    for word in words:
        prefixes=trie.getAllPrefixesOfWord(word)
        for prefix in prefixes:
            queue.append( (word, word[len(prefix):]) )
        trie.insert(word)

    #Process the queue
    longestWord=''
    maxLength=0
    while queue:
        word, suffix = queue.popleft()
        if suffix in trie and len(word)>maxLength:
            longestWord=word
            maxLength=len(word)
        else:
            prefixes=trie.getAllPrefixesOfWord(suffix)
            for prefix in prefixes:
                queue.append( (word, suffix[len(prefix):]) )
```

```
    return longestWord
```

The complexity of this algorithm is O(kN) where N is the number of words in the input list, and k the maximum number of words in a compound word. The number k may vary from one list to another, but it'll generally be a constant number like 5 or 10. So, the algorithm is linear in number of words in the list, which is an optimal solution to the problem.