**Tree Isomorphism Problem**

Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

For example, following two trees are isomorphic with following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.

/\* Given a binary tree, print its nodes in reverse level order \*/

bool isIsomorphic(node\* n1, node \*n2)

{

// Both roots are NULL, trees isomorphic by definition

if (n1 == NULL && n2 == NULL)

return true;

// Exactly one of the n1 and n2 is NULL, trees not isomorphic

if (n1 == NULL || n2 == NULL)

return false;

if (n1->data != n2->data)

return false;

// There are two possible cases for n1 and n2 to be isomorphic

// Case 1: The subtrees rooted at these nodes have NOT been "Flipped".

// Both of these subtrees have to be isomorphic, hence the &&

// Case 2: The subtrees rooted at these nodes have been "Flipped"

return

(isIsomorphic(n1->left,n2->left) && isIsomorphic(n1->right,n2->right))||

(isIsomorphic(n1->left,n2->right) && isIsomorphic(n1->right,n2->left));

}

**Reverse Level Order Traversal**

We have discussed [level order traversal](http://www.geeksforgeeks.org/level-order-tree-traversal/) of a post in previous post. The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.

Example Tree

Reverse Level order traversal of the above tree is “4 5 2 3 1″.

Soln:

* Use normal BFS logic using queue.
* While popping up the element from the queue, push that element into stack.
* After completion of BFS, pop all the elements from stack and print it.
* Printing order can be easily changed by changing the direction of traversing in BFS.

i.e. 4 5 2 3 1 and 5 4 3 2 1 ... The only difference is in the pushing order of children in the queue.

**Find if there is a triplet in a Balanced BST that adds to zero**

Expected time complexity is O(n^2) and only O(Logn) extra space can be used.

You can modify given Binary Search Tree.

Note that height of a Balanced BST is always O(Logn)  
For example, isTripletPresent() should return true for following BST because there is a triplet with sum 0, the triplet is {-13, 6, 7}.

Solution:

Change the BST to DLL.

Use the logic of finding the triplet from sorted array which has TC O(n^2).

If found then print the triplet.

So here converting BST to DLL takes O(n) TC and O(logn) SC

then O(n^2) TC to find the triplet...

Nice problem... :) ... enjoy...

**Decision Trees – Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)**

Let us solve the classic “fake coin” puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume N = 8.

***Easy:*** *Given a two pan fair balance and N identically looking coins, out of which only one coin is* ***lighter (or heavier)****. To figure out the odd coin, how many minimum number of weighing are required in the worst case?*

***Difficult:*** *Given a two pan fair balance and N identically looking coins out of which only one coin* ***may be*** *defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?*

Let us start with relatively simple examples. After reading every problem try to solve on your own.

**Problem 1: (Easy)**

*Given 5 coins out of which one coin is* ***lighter****. In the worst case, how many minimum number of weighing are required to figure out the odd coin?*

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin in heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

**Analysis:** In general, if we know that the coin is heavy or light, we can trace the coin in log3(N) trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k-th level will have 3k leaves and hence we need 3k >= N.

In other-words, in ***k*** trials we can examine upto ***3k*** coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because 32 < 12 < 33.

**Problem 2: (Difficult)**

*We are given 4 coins, out of which only one coin* ***may be*** *defective. We don’t know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.*

From the above analysis we may think that k = 2 trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than N = 4 (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.

The outcome can be (12) < (34) i.e. we go on to left subtree or (12) > (34) i.e. we go on to right subtree.

The left subtree is possible in two ways,

* A) Either 1 or 2 can be lighter OR
* B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh (1, 2) or (3, 4). Let us consider (3, 4) as the analogy for (1, 2) is similar. The outcome of second trail can be three ways

* A) (3) < (4) yielding 4 as defective heavier coin, OR
* B) (3) > (4) yielding 3 as defective heavier coin OR
* C) (3) = (4), yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took (3, 2) where 3 is confirmed as genuine. We can get (3) > (2) in which 2 is lighter, or (3) = (2) in which 1 is lighter. Note that it impossible to get (3) < (2), it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

**Analysis:** Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast (2N + 1) leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total (2N + 1) leaves.

As explained earlier ternary tree at level k, can have utmost 3k leaves and we need a tree with leaves of 3k > (2N + 1).

*In other words, we need atleast k > log3(2N + 1) weighing to find the defective one.*

Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is going to yield valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

**Problem 3: (Special case of two pan balance)**

*We are given 5 coins, a group of 4 coins out of which one coin is defective (we* ***don’t know*** *whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?*

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as [(G1, 23) and (4)]. Any other group can’t generate full 3-ary tree, try yourself. The following diagram explains the procedure.

The middle case (G1) = (23) is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case (G1) < (23). This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance could be (2) < (3) yielding 3 as heavier or (2) > (3) yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case (G1) > (23). This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case could be (2) < (3) yielding 2 as lighter coin, or (2) > (3) yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with (N + 1) = 5 coins where N = 4, we end up with (2N + 1) = 9 leaves. ***Infact we should have 11 outcomes since we stared with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure these two out comes?***

If we observe the figure, after the first weighing the problem reduced to “we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)”. This can be solved in one weighing (read Problem 1).

**Analysis:** Given (N + 1) coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of (2N + 1) leaves. Since the total possible outcomes are (2(N + 1) + 1), number of weighing (trials) are given by the height of ternary tree, k >= log3[2(N + 1) + 1]. *Note the equality sign*.

Rearranging k and N, *we can weigh maximum of N <= (3k – 3)/2 coins in k trials*.

**Problem 4: (The classic 12 coin puzzle)**

*You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?*

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can’t lead to best tree. *From the above two examples, we can ensure that the decision tree can be used in optimal way if we can reveal atleaset one genuine coin*. Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, … 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

* (1234) = (5678), both groups are equal. Defective coin may be in (ABCD) group.
* (1234) < (5678), i.e. first group is less in weight than second group.
* (1234) > (5678), i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where (1234) < (5678). It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8 ) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing, [can you try](http://geeksforgeeks.org/forum/topic/odd-ball-out)?). If we weigh these two groups again the outcome can be three ways, i) (1235) < (4BCD) yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) (1235) = (4BCD) yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) (1235) > (4BCD) yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where (1234) > (5678)) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing in the worst case.

**Few Interesting Puzzles:**

* Solve Problem 4 with N = 8 and N = 13, How many minimum trials are required in each case?
* Given a function *int weigh(A[], B[])* where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if A < B and 1 if A > B. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of others, smaller or greater than others. Write a program to find the odd element (if any) using *weigh()* minimum number of times.
* You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are needed to figure out odd coin?

**References:**

Similar problem was provided in one of the exercises of the book ”Introduction to Algorithms by Levitin”. Specifically read section 5.5 and section 11.2 including exercises.

**Tournament Tree (Winner Tree) and Binary Heap**

March 23, 2011

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

[Tournament tree](http://en.wikipedia.org/wiki/Selection_algorithm#Tournament_Algorithm) is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](http://geeksforgeeks.org/?p=8870) (put n = 2 in equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

**Second Best Player**

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in **(N + log2N – 2)** comparisons. For details read [this comment](http://www.geeksforgeeks.org/archives/4184/comment-page-1#comment-2541).

The following diagram displays a  tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.

The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

**Median of Sorted Arrays**

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL (log2M)** to have atleast M external nodes.

Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

{ 2, 5, 7, 11, 15 } ---- Array1

{1, 3, 4} ---- Array2

{6, 8, 12, 13, 14} ---- Array3

What should be the height of tournament tree? We need to construct a tournament tree of height log23 .= 1.585 = 2 rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.

After the first tournament, the tree appears as below,

We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

*Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.*

After the second tournament, the tree appears as below,

The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L1, L2 … Lm requires time complexity of ***O((L1 + L2 + … + Lm) \* logM)*** to merge all the arrays, and ***O(m\*logM)*** time to find median, where ***m*** is median position.

**Select smallest one million elements from one billion unsorted elements:** [Read the Source](http://geeksforgeeks.org/forum/topic/google-interview-question-for-software-engineerdeveloper-fresher-3).

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since 29 < 1000 < 210, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

**Implementation**

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2k-1 to 2k – 1 where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

**Related Posts**

[Link 1](http://geeksforgeeks.org/forum/topic/kth-smallest-element-in-the-union-of-the-arrays-in-a-logarithmic-time-algorithm), [Link 2](http://www.geeksforgeeks.org/archives/4184/comment-page-1#comment-2541), [Link 3](http://geeksforgeeks.org/forum/topic/kth-smallest-element-in-the-union-of-the-arrays-in-a-logarithmic-time-algorithm), [Link 4](http://geeksforgeeks.org/forum/topic/quick-sort-in-worst-case-onlogn-1), [Link 5](http://geeksforgeeks.org/forum/topic/google-interview-question-for-software-engineerdeveloper-about-algorithms-arrays), [Link 6](http://geeksforgeeks.org/forum/topic/yahoo-interview-question-about-arrays), [Link 7](http://geeksforgeeks.org/forum/topic/fining-the-median), [Link 8](http://geeksforgeeks.org/?p=2105).

**Pattern Searching | Set 8 (Suffix Tree Introduction)**

Given a text txt[0..n-1] and a pattern pat[0..m-1], write a function search(char pat[], char txt[]) that prints all occurrences of pat[] in txt[]. You may assume that n > m.

We will preprocesses the text.

A suffix tree is built of the text.

After preprocessing text (building suffix tree of text), we can search any pattern in O(m) time where m is length of the pattern.  
Imagine you have stored complete work of [William Shakespeare](http://en.wikipedia.org/wiki/William_Shakespeare) and preprocessed it. You can search any string in the complete work in time just proportional to length of the pattern. This is really a great improvement because length of pattern is generally much smaller than text.  
Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

**A Suffix Tree for a given text is a compressed trie for all suffixes of the given text**. We have discussed [Standard Trie](http://www.geeksforgeeks.org/trie-insert-and-search/). Let us understand **Compressed Trie** with the following array of words.

{bear, bell, bid, bull, buy, sell, stock, stop}

Following is standard trie for the above input set of words.

Following is the compressed trie. Compress Trie is obtained from standard trie by joining chains of single nodes. The nodes of a compressed trie can be stored by storing index ranges at the nodes.

**How to build a Suffix Tree for a given text?**  
As discussed above, Suffix Tree is compressed trie of all suffixes, so following are very abstract steps to build a suffix tree from given text.  
1) Generate all suffixes of given text.  
2) Consider all suffixes as individual words and build a compressed trie.

Let us consider an example text “banana\0″ where ‘\0′ is string termination character. Following are all suffixes of “banana\0″

banana\0

anana\0

nana\0

ana\0

na\0

a\0

\0

If we consider all of the above suffixes as individual words and build a trie, we get following.

If we join chains of single nodes, we get the following compressed trie, which is the Suffix Tree for given text “banana\0″

Please note that above steps are just to manually create a Suffix Tree. We will be discussing actual algorithm and implementation in a separate post.

**How to search a pattern in the built suffix tree?**  
We have discussed above how to build a Suffix Tree which is needed as a preprocessing step in pattern searching. Following are abstract steps to search a pattern in the built Suffix Tree.  
**1)** Starting from the first character of the pattern and root of Suffix Tree, do following for every character.  
…..**a)** For the current character of pattern, if there is an edge from the current node of suffix tree, follow the edge.  
…..**b)** If there is no edge, print “pattern doesn’t exist in text” and return.  
**2)** If all characters of pattern have been processed, i.e., there is a path from root for characters of the given pattern, then print “Pattern found”.

Let us consider the example pattern as “nan” to see the searching process. Following diagram shows the path followed for searching “nan” or “nana”.

**How does this work?**  
Every pattern that is present in text (or we can say every substring of text) must be a prefix of one of all possible suffixes. The statement seems complicated, but it is a simple statement, we just need to take an example to check validity of it.

**Applications of Suffix Tree**  
Suffix tree can be used for a wide range of problems. Following are some famous problems where Suffix Trees provide optimal time complexity solution.  
1) Pattern Searching  
2) [Finding the longest repeated substring](http://en.wikipedia.org/wiki/Longest_repeated_substring_problem)  
3) [Finding the longest common substring](http://en.wikipedia.org/wiki/Longest_common_substring_problem)  
4) [Finding the longest palindrome in a string](http://en.wikipedia.org/wiki/Longest_palindromic_substring)

There are many more applications. See [this](http://en.wikipedia.org/wiki/Suffix_tree#Functionality) for more details.

We will soon be writing separate posts for detailed algorithms for building suffix trees, time complexity and implementation.

|  |
| --- |
|  |

**References:**  
<http://fbim.fh-regensburg.de/~saj39122/sal/skript/progr/pr45102/Tries.pdf>  
<http://www.cs.ucf.edu/~shzhang/Combio12/lec3.pdf>  
<http://www.allisons.org/ll/AlgDS/Tree/Suffix/>