Q1. Given-a-binarytree-and-a-sum-PRINT-all-paths-where-sum-equals-given-sum

Total number of paths from root to leaf == number of leaves == 2^(log n -1)

Time complexity to find all these == O(n)

But you also have to print. There are O(log n) nodes in a path, So time complexity of printing == O(n logn)

http://stackoverflow.com/questions/24601111/whats-time-complexity-of-this-algorithm-for-finding-all-path-sum

Q2. Total nodes in a binary tree

level nodes

0 0 1

1 0 0 2

2 0 0 0 0 4

So, 2^(log n) - 1

Another way to look at it

1 + 2 + 4 + ... + 2^(logn-1)

2^0+2^1+2^2+ ... + 2^(logn-1)

1 + r + r2 + r3 + ... r^n-1 = a(1-r^n)/(1-r)

= (1-2^logn)/(1-2)

= 2^logn - 1

http://math.stackexchange.com/questions/141783/the-maximum-number-of-nodes-in-a-binary-tree-of-depth-k-is-2k-1-k-geq1

Q3. Derivation for Geometric series

Sn = sigmak=0 to n-1 r^k = r^0 + r^1 + r^2 + r^3 + ... + r^n-1

rSn= = r^1 + r^2 + r^3 + r^4 + ... + r^n

rSn - Sn = r^n - 1

Sn(r-1) = r^n - 1

Sn = (r^n - 1)/(r-1)

Sn (n=~) = (r^~ - 1)/(r-1) = 1/1-r

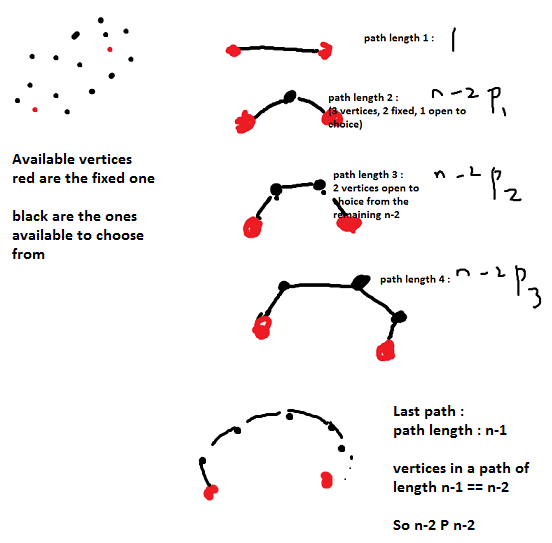
Q4. Paths in a tree and a graph

1. **Number of paths between any two nodes in a graph**

Disclaimer : Within a path, we don’t go back to the same node again, even if you see the code, once a node is marked visited for a path, we don’t visit it again, unless we reach the end of the path, that is, all nodes marked visited

Follow the code gos\src\Graphs\minimummaximum\_hops\_between\_2nodes\Backtracking.java and BacktrackingExplicit.java because these are the ones who need to compute all paths, and compare among them to find the minimum/ BFS always works in O(V+E)

Time complexity of finding all paths = n! factorial, like I told ‘vish’ from slack, thinking by myself. Actually, I told him n^n exponential, but as the disclaimer says, we don’t go back to a previously visited node again, otherwise, you can just keep circling infinite times, as Vivek said during the `worst meetup | Chess game | Sameer BFS | Supriya missed`. So, it’s n factorial, but here , is a more formal proof by vish.



So sum == (n-2)P(0) + (n-2)P(1) + (n-2)P(2) + … + (n-2)P(n-2)

Adding all that up, 1 + n-2P1+n-2P2+. n-2Pn-2 is the greatest integer lower than (n-2)! \* e.

Proof [here](http://math.stackexchange.com/questions/161314/what-is-the-sum-of-following-permutation-series-np0-np1-np2-cdots-npn/161317)

1. **Number of paths between any node to any other node in a graph ()**

Disclaimer : Within a path, we don’t go back to the same node again, even if you see the code, once a node is marked visited for a path, we don’t visit it again, unless we reach the end of the path, that is, all nodes marked visited

(n)P(1) + (n)P(2) + (n)P(3) + … + (n)P(n) = n!

Proof [here](http://math.stackexchange.com/questions/161314/what-is-the-sum-of-following-permutation-series-np0-np1-np2-cdots-npn/161317)

1. **Number of paths between root-to-leaf in a binary tree**There are as many paths as the number of leaf nodes, which is 2^( (logn) -1)

But to find all these paths, the time required is O(n) since you would have to visit all nodes

1. **Number of paths between any two nodes in a binary tree**

2 ^ n

Q5. Concatenating a list/array of n strings time complexity

public String joinWords(String[] words){

String sentence = “”;  
 for(String w : words)  
 {  
 sentence = sentence + w;

}

}  
In every iteration, it makes a new copy

The first iteration requires you to copy x characters, the second 2x, the third 3x….. nx == 1+2+3+…n == O(n^2)

Q6. Time complexity of recursive methods in general  
== (Number of calls in this frame)^number of frames (if number of calls >= 2)  
== Linear number of calls (Basically, number of frames) (if no. of calls ==1)   
How does this formula work – If it’s 2 calls in a frame, just think of it as splitting in to two, and it has to reach the end (that is, base case) for each path.

Say Sudoku, in every frame, you make 9 calls. There can be upto 81 frames. So time complexity = 9^81

Q7.