DSA masterclass

Become a DSA expert + a good problem solver. Tips from experts

30dayscoding.com

Watch the videos

https://youtube.com/playlist?list=PLxbAS7NVaSZIgqpVNyOA0_ftJN9i1Mls_&feature=shared

Important points

1. DSA is about problem solving and not just leetcode
2. Use DSA as your advantage instead of complaining
3. No need for CP (right now) if you are not confident with 250+ questions
4. No need of a course if you know the basics
5. Stop pasting solutions and watching videos without TRYING
6. Stop comparing your journey to others
7. No, you don’t need to solve 1000 questions
8. No, you don’t have to be a born genius

I cleared 40+ DSA interviews and you can too!

Table of Contents

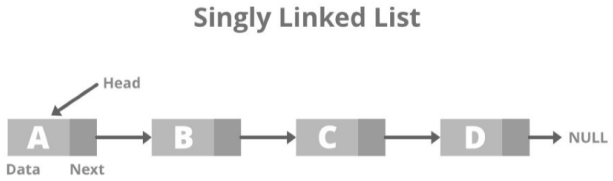
The document is divided as follows.

Linked List

Introduction

⋆ Linked list is a data structure which stores objects in nodes in a snake-like structure. Instead of an array (where we have a simple list to store something) we have nodes in the linked list. It's the same thing though, you can store whatever you want - objects, numbers, strings, etc.

The only difference is in the way it's represented. It's like a snake: with a head and tail, and you can only access one thing at a time - giving its own advantages and disadvantages. So if you want to access the 5th thing, you can't do linked_list[5], instead -> you would have to iterate over the list from the beginning and then stop when the number hits 5.



Problem 1: Linked list methods

Here's how a linked list looks like: Linked list: Methods

Arrays

Introduction

⭐ Informally, an array is a list of things. It doesn't matter what the things are; they can be numbers, words, apple trees, or other arrays. Each thing in an array is called an *item* or *element.* Usually, arrays are enclosed in brackets with each item separated by commas, like this: [1, 2, 3]. The elements of [1, 2, 3] are 1, 2, and 3.

- Introduction to Arrays
- https://www.cs.cmu.edu/~15122/handouts/03-arrays.pdf
- An Overview of Arrays and Memory (Data Structures & Algorithms #2)
- What is an Array? - Processing Tutorial

Arrays are used with all different types of data structures to solve different problems, so it's kind of hard to come up with array questions with just an array logic. Let's discuss some of the most famous patterns which concern arrays most of the time.

2D matrices are also arrays and are very commonly asked in interviews. A lot of graph, DP, and search based questions involve the use of a 2D matrix and it's important to understand the core concepts there. We've discussed the common patterns in each section below so make sure to check that out.

Hash maps, tables

⭐ A hash table is a data structure that implements an associative array abstract data type, a structure that can map keys to values. In other words, we can store anything in the form of key value pairs.
Example: map<string, string>, means that this is a hashmap where we store string key and value pairs.

Problem 4: Merge sorted lists

Merge Two Sorted Lists

⭐ We have 2 sorted lists and we want to merge them into one.

Does sorting tell you something? The element at the head would be the smallest.

Can we compare the heads every time and add those to the new list?

Ooooo, maybe yeah. Let's try comparing and then move the counter of the bigger element.

```
if l1.val < l2.val:
      cur.next = l1
      l1 = l1.next
```

Otherwise, we do this for the other node (because that's smaller)

```
else:
    cur.next = l2
    l2 = l2.next
```

What do we do once a list is done and the other one is left? Simply move the new linked list to the next pointer -> cur = cur.next and then add all the elements of the left over list.

```
cur.next = l1 or l2 # iterate over and add all the elements
return head (the temporary new list that we made)
```

Problem 5: Merge K Sorted lists:

Leetcode 23. Merge k Sorted Lists

⭐ We have k lists and we want to merge all of those into a big one.

1. One simple way would be to compare every 2 lists, call this function, and keep doing until we have a bigger list. Any other shorter way?
2. We can be smart about it and add all the lists into one big array or list -> sort the array -> then put the elements back in a new linked list!

Read

- Linked list: Methods
- How I Taught Myself Linked Lists. Breaking down the definition of linked list
- Introduction to Linked List

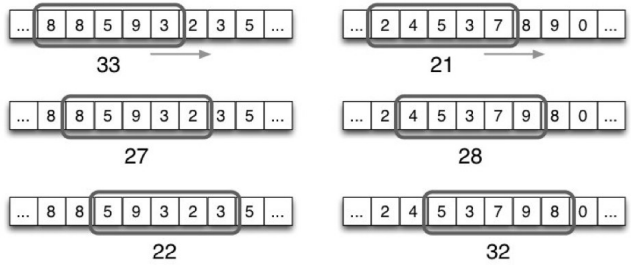
Videos

- Data Structures: Linked Lists
- Interview Question: Nth-to-last Linked List Element

Questions

- 141. Linked List Cycle (Leetcode)
- Delete Node in a Linked List"
- 19. Remove Nth Node From End of List
- Merge Two Sorted Lists
- Palindrome Linked List
- 141. Linked List Cycle (Leetcode)
- Intersection of Two Linked Lists
- Remove Linked List Elements
- Middle of the Linked List
- lc 23. Merge k Sorted Lists
-

the string/array.



Sliding window is a 2 pointer problem where the front pointer explores the array and the back pointer closes in on the window. Here's an awesome visualization to understand it more:
Dynamic Programming - Sliding Window

Problem 1: Max sum for consecutive k

⋆ We have an array [1,2,3,2,4] and k=2, we want to return the max sum of the array with size 2.
Looking at this for the first time, I would think of a brute force way to calculate all the subarrays, find their sum, store the maximum, and return it.
However, that's very expensive. We don't really need to explore all the subarrays. Or, we can do that in an easier way (which is also cheaper): SLIDING WINDOW.

This is how sliding window would work here:

- We start with a window of 'k' from the left.
- We plan to move it to the right until the very end
- We remove the leftmost element (from the window) and add the right one as we move to the left
- We store the sum for every window and then return the max at the very end.

Storing the sum

- You can either calculate the sum every time -> which will be expensive
- Or we can just find the sum of the window the first time

Binary Search

Introduction

⋆ We use binary search to optimize our search time complexity when the array is sorted (min, max) and has a definite space. It has some really useful implementations, with some of the top companies still asking questions from this domain.

The concept is: if the array is sorted, then finding an element shouldn't require us to iterate over every element where the cost is O(N). We can skip some elements and find the element in O(logn) time.

Algorithm

⋆ We start with 2 pointers by keeping a low and high -> finding the mid and then comparing that with the number we want to find. If the target number is bigger, we move right -> as we know the array is **sorted**. If it's smaller, we move left because it can't be on the right side, where all the numbers are bigger than the mid value.

Here's an iterative way to write the Binary search algorithm:

```
int left = 0, right = A.length - 1;
// loop till the search space is exhausted
while (left <= right)
{
```

Problem 1: Max font size (Google internship)

⋆ Google likes to test you on word problems with core principles. So even if they ask you a binary search question, it will be framed like a real life thing so that it's much harder to understand. They also test OOPS sometimes, by asking you to create classes and functions to display different things. Here's the question:

Given:

1. Height and width of a screen where you have to type
2. Height and width of each character you type on the screen
3. Min and max range of the the font size of each character

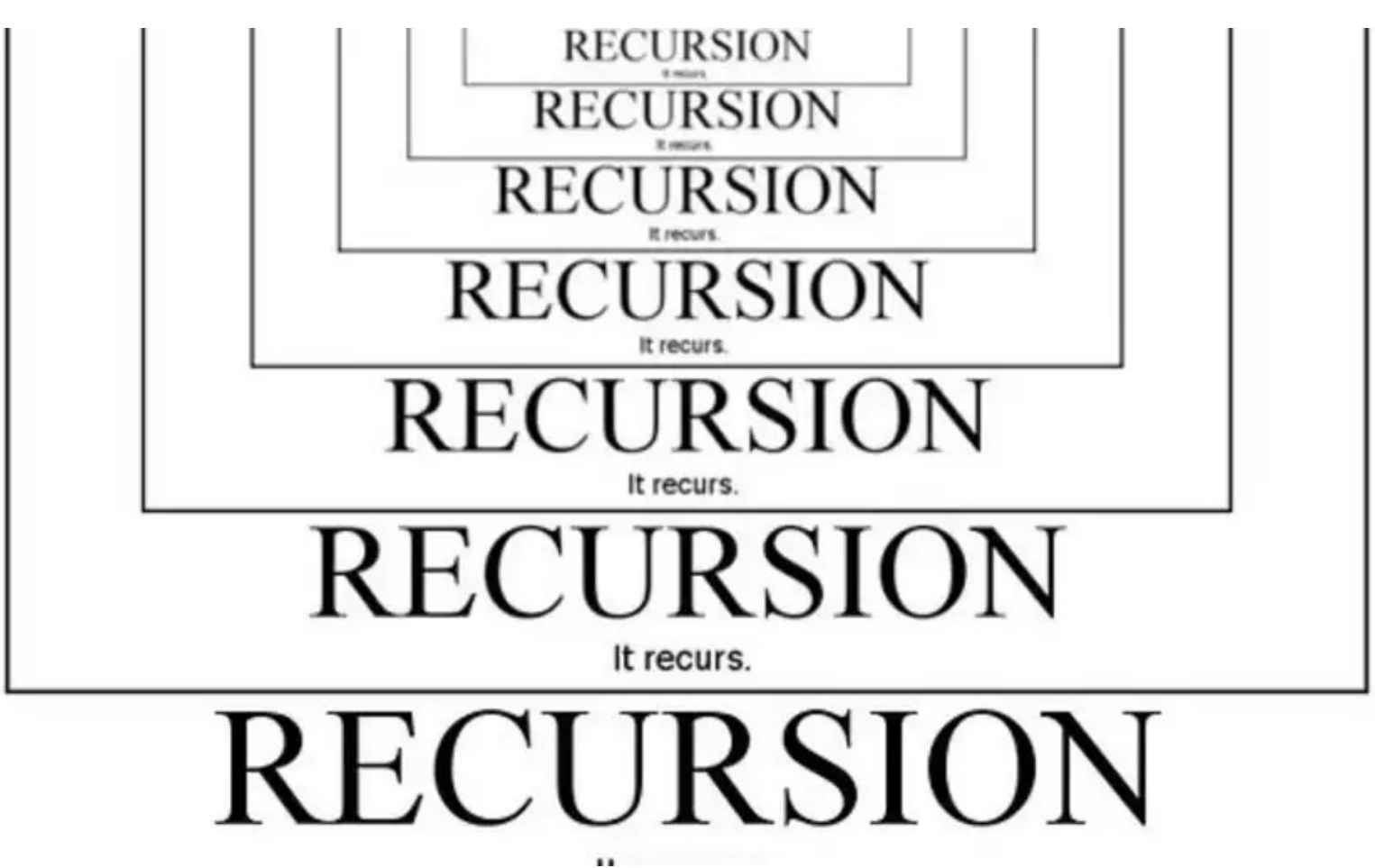
Find the maximum font size such that the characters fit inside the screen

Once you understand the question, it's trivial to think of a brute force problem: explore all the possible font sizes and then see what fits at the end. Return that. Thinking a little more, we see that we have a range (sorted) and we don't really have to check for each font before choosing the maximum one. Shoot -> it's binary search.

Here's how the pseudo code looks like:

```
def find_max_font():
    max_font = 0
    start, end = min_font, max_font
    while start<=end:
        mid_font = start + (end-start)//2
```

BINARY SEARCH


RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION
It recurs.
RECURSION

⭐ These are some questions I have when I look at a recursive question/solution, you probably have the same. Let's try to figure out them

- What happens when the function is called in the **middle** of the whole recursive function?
- What happens to the stuff **below** it?
- What do we think of the base case?
- How do we figure out when to **return** ?
- How do we save the value, specially in the **true/false** questions?
- How does **backtracking** come into place, wrt recursion?

Let's try to answer these one by one. A recursive function means that we're breaking the problem down into a smaller one. So if we're saying function(x/2) -> we're basically calling the function again with the same parameters.
So if there's something below the recursive function -> that works with the same parameter. For instance, calling function(x/2) with x=10 and then printing (x) after that would print 10 and then 5 and so on. Think of it as going back to the top of the function, but with different parameters.

The return statements are tricky with recursive functions. You can study about those things, but practice will help you get over it. For instance, you have fibonacci, where we want to return the sum of the last 2 elements for the current element -> the code is something like *fib(n) + fib(n-1)* where fib() is the recursive function. So this is solving the smaller problem until when? -> Until the base case. And the base case will return 1 -> because eventually we want the fib(n) to return a number. This is a basic example, but it helps you gain some insights on the recursive part of it.

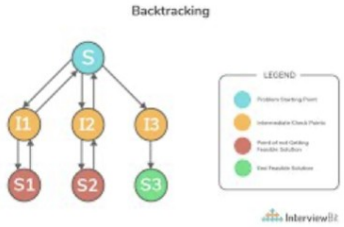
Something complex like dfs or something doesn't really return anything but transforms the 2d matrix or the graph.

RECURSION

- V v v important topic
- Read every line
- Understand every pattern

This is the basis for backtracking and DP - 2 very important topics

- Don't skip questions
- Make notes
- Solve 25+ problems



Problem 1: Generate parentheses

22. Generate Parentheses

⋆ Generate balanced parentheses, given a number.

In simple words, we want to print out all the possible cases -> valid parentheses can be generated.

One thing which strikes me is -> we need a way to add “(” and “)” to all possible cases and then find a way to validate so that we don’t generate the unnecessary ones.

The first condition is if there are more than 0 open / left brackets, we recurse with the right ones. And if we have more than 0 right brackets, we recurse with the left ones. Left and right are **initialized** at N - the number given.

Problem 3: Letter combination of phone numbers

17. Letter Combinations of a Phone Number

⋆ Interesting problem and can be solved both iteratively and recursively (same for any pr

The first thing which comes to mind is to have a map of the numbers and digits, so that w

actually use it. The second thing which is trivial is that -> we would iterate over, take all th

possible ways, and then store it in a list. It’s basically a cliche backtracking problem where

have some arrays and we want all the possible cases in those.

A recursive function would need to have something in the arguments which we add + we

the array (using python sub-array)

```
combo(combination+letter, digits[1:])
```

We do this for every letter and add a base case for adding the combination to the result a

Here’s how the complete code looks like

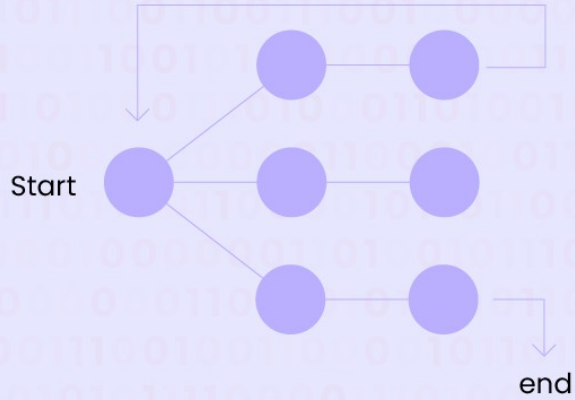
```
def combo(combination, digits):
      if len(digits)==0:
          a.append(combination)

      else:
          for letter in phone[digits[0]]:
              combo(combination+letter, digits[1:])
```

Here’s a java solution code for it: My recursive solution using Java

READ AND UNDERSTAND PATTERNS.

- Copy pasting doesn’t help
- Reading doesn’t help
- Understanding helps

BACKTRACKING



Introduction

⭐ Backtracking can be seen as an optimized way to brute force. Brute force approaches evaluate every possibility. In backtracking you stop evaluating a possibility as soon as it breaks some constraint provided in the problem, take a step back and keep trying other possible cases, see if those lead to a valid solution.

Think of backtracking as exploring all options out there, for the solution. You visit a place, there's nothing after that, so you just come back and visit other places. Here's a nice way to think of any problem:

- Recognize the pattern
- Think of a human way to solve it
- Convert it into code.

Problem 1: Permutations

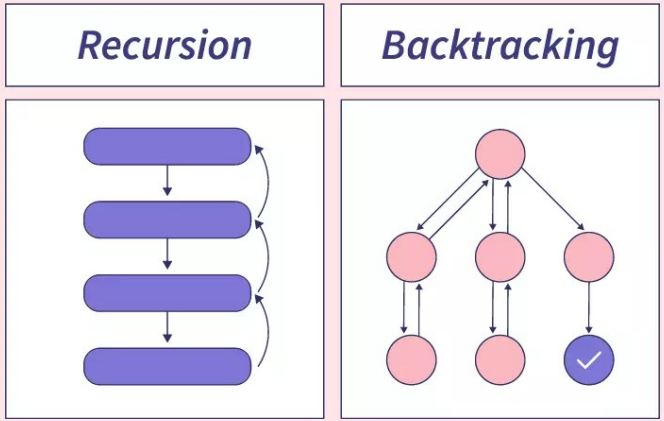
46. Permutations

⭐ We have an array [1,2,3] and we want to print all the possible permutations of this array. The initial reaction to this is - explore all possible ways -> somehow write 2,1,3, 3,1,2 and other permutations.

Second step, we recognize that there's a pattern here. We can start from the left - add the first element, and then explore all the other things with the rest of the items. So we choose 1 -> then add 2,3 and 3,2 -> making it [1,2,3] and [1,3,2]. We follow the same pattern with others.

How do we convert this into code?

- Base case
- Create a temporary list
- Iterate over the original list
    - Add an item + mark them visited
    - Call the recursive function
    - Remove the item + mark them univisited



Problem 3: Combination Sum

39. Combination Sum

⭐ We want to return the numbers which would add up to the target number given. We have to return all the possible combinations. So this is basically all subsets (with repeats allowed) with a target given.

From the get go, I know one thing -> we want to explore all cases, find the ones where the target matches, and then add that to a list, and return that list.

Backtracking template: Make a choice

- Iterate over the array
  - Add the item
  - Backtrack - recursive call
  - Remove the item

```
for(int i = start; i < nums.length; i++){
    tempList.add(nums[i]);
    backtrack(list, tempList, nums, target_left - nums[i], i); // not i + 1 because we can reuse same elements
    tempList.remove(tempList.size() - 1);
}
```


A good thing to note here is that we pass in the target_left - nums[i] which basically means that we're choosing that element and then subtracting that from what we have in the argument. So the base case with this would be

Target_left == 0 -> because that's when we know we can make the target.

One other thing to save some time and memory can be target_left < 0 -> to return when we reach here, because negative numbers can never become positive numbers. So once the target_left is below 0, it can never come up -> good to just return;

```
public List<List<Integer>> combinationSum(int[] nums, int target) {
```

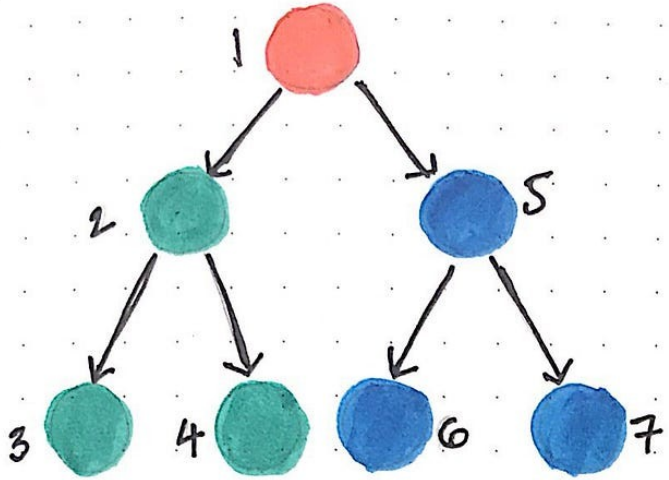
Backtracking is v v important pattern / algo / etc

- Used in real life use cases
- Finding cheapest flights/shoes/etc
- Finding fastest etc
- UNDERSTAND THE PATTERN HERE

Push, recurse, pop is a famous pattern

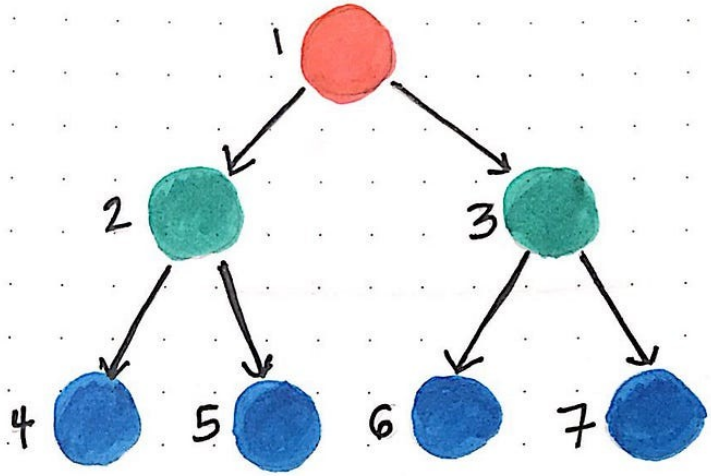
- We use the element
- Recurse and solve for it
- Then remove it

Read every word and line from this.



Depth-first search

- Traverse through left subtree(s) first, then traverse through the right subtree(s).



Breadth-first search

- Traverse through one level of children nodes, then traverse through the level of grandchildren nodes (and so on...).

Here's a beautiful visualization of a search in a tree: Branch and Bound - Depth-Limited Search
Here's a general iterative dfs pseudo-code template:

```
def dfs(root, target):
    stack = []
    stack.append(root) # add the first item

    while len(stack)>0:
        node = stack.pop() # pop the grid item

        if(node == target):
            return true

        # explore more
        # For trees -> if root.left or root.right
        if (condition):
            stack.append(new_item)

    return false;
```

The second step is that of MEMOIZATION and we want to keep a track of all the nodes visited when we're iterating over. Here's a complete version of a BFS algorithm where we keep track of the visited node using an array **discovered []**
This could be anything - array, map, set - depending on the situation. The only thing we need is to store the visited things so that we're not repeating any work.

```
public static void BFS(Graph graph, int v, boolean[] discovered)
    {
        // create a queue for doing BFS
        Queue<Integer> q = new ArrayDeque<>();
```

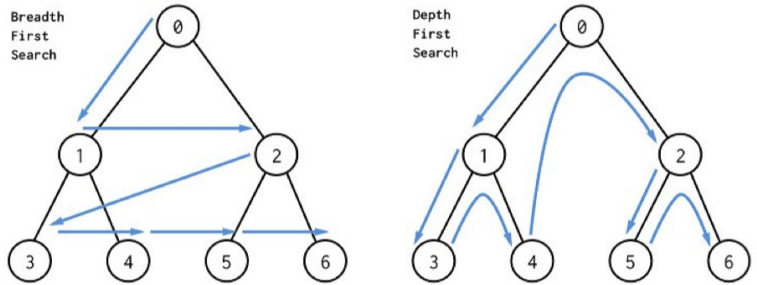

Here are some implementations and use cases for DFS, BFS:
DFS:

- Find connected components in a graph
- Calculate the vertex or edges in a graph
- Whether the graph is strongly connected or not
- Wherever you want to explore everything or maybe go in depth

BFS

- Shortest path algorithms and questions
- Ford fulkerson algorithm
- Finding nodes in a graph
- Wherever there is a shortest thing, finding something quickly, etc.

BlockTrain.info



```java
public int numIslands(char[][] grid) {
        int count=0;
        for(int i=0;i<grid.length;i++){
            for(int j=0;j<grid[0].length;j++){
                if(grid[i][j] == '1'){
                    dfs(grid, i, j);
                    count+=1;
                }
            }
        }
        return count;
    }

    public void dfs(char[][] grid, int i, int j){
        if(i<0 || i>=grid.length || j<0 || j>=grid[0].length){
            return;
        }
        if(grid[i][j]== '1'){
            grid[i][j] = '#';
            dfs(grid, i+1,j);
            dfs(grid, i,j+1);
            dfs(grid, i,j-1);
            dfs(grid, i-1,j);
        }
    }
```

The most important DFS problem - number of islands.

- Use a pen and paper
- Make a diagram
- Visualize how things are working

Solution

- Iterate, change the number
- Count the islands
- Change it back
- Do it for all the indexes

```
count=0
for i in range(len(grid)):
    for j in range(len(grid[0])):
        if grid[i][j]=='1':
            dfs(grid, i, j)
            count+=1
return count

def dfs(grid, i, j):
    s=[]
    s.append((i,j))
    while len(s)>0:
        a,b = s.pop()
        grid[a][b]='X'
        if a>0 and grid[a-1][b]=='1':
            s.append((a-1,b))
        if b>0 and grid[a][b-1]=='1':
            s.append((a,b-1))
        if a<len(grid)-1 and grid[a+1][b]=='1':
            s.append((a+1,b))
        if b<len(grid[0])-1 and grid[a][b+1]=='1':
            s.append((a,b+1))
```

```
public int numIslands(char[][] grid) {
        int count=0;
        for(int i=0;i<grid.length;i++){
            for(int j=0;j<grid[0].length;j++){
                if(grid[i][j] == '1'){
                    dfs(grid, i, j);
                    count+=1;
                }
            }
        }
        return count;
    }

    public void dfs(char[][] grid, int i, int j){
        if(i<0 || i>=grid.length || j<0 || j>=grid[0].length){
            return;
        }
        if(grid[i][j]== '1'){
            grid[i][j] = '#';
            dfs(grid, i+1,j);
            dfs(grid, i,j+1);
            dfs(grid, i,j-1);
            dfs(grid, i-1,j);
        }
    }
```

ITERATIVE VS RECURSIVE

SOLVE USING BOTH

Problem 3: Rotten oranges

994. Rotting Oranges

⭐ Every minute a fresh orange turns rotten if it's around a rotten orange. Similar to life -> if you're around negative people, you tend to be negative. Keep a positive outlook, help everyone, and take things forward!

This is an amazing question -> let's understand the iterative way of doing this and how to solve any searching related question with a stack or queue -> iteratively. We have the minimum condition here, so using BFS is the way to go! A simple pattern, as discussed before is:

- Prepare the stack/queue -> Add the initial nodes
- Pop the node from stack, mark it visited, add the valid neighbors
- Repeat the process for the new nodes.

First step is to prepare the queue. We add the rotten oranges (represented by 2) to the queue and also count the total number of oranges. 0 -> means an empty place.

```
for (int i = 0; i < grid.length; i++) {
    for (int j = 0; j < grid[0].length; j++) {
        if (grid[i][j] != 0) total++;
        if (grid[i][j] == 2) q.offer(new Pair(i, j));
    }
}
```

We have the queue ready and now we iterate until it's empty: *while (stack.isEmpty()) {}*. We want to add all the neighbors of the current orange, which are in 4 directions and here's something to note when you have conditions like this.

When we want to traverse in all 4 directions, or maybe in 8 directions if we have a double condition, we can make a directions dictionary and iterate over it. Something like: [[0,1], [0,-1], [1,1], [1, 0]] or *int[][] dirs = {{1,0},{-1,0},{0,1},{0,-1}};*

```
while (! q.isEmpty()) {
    int size = q.size();
    rotten += size;
    // if the total number of rotten oranges matches our local variable
    // then return the time it took
    if (rotten == total_rotten) return time;
```

Read

- Leetcode patterns 1

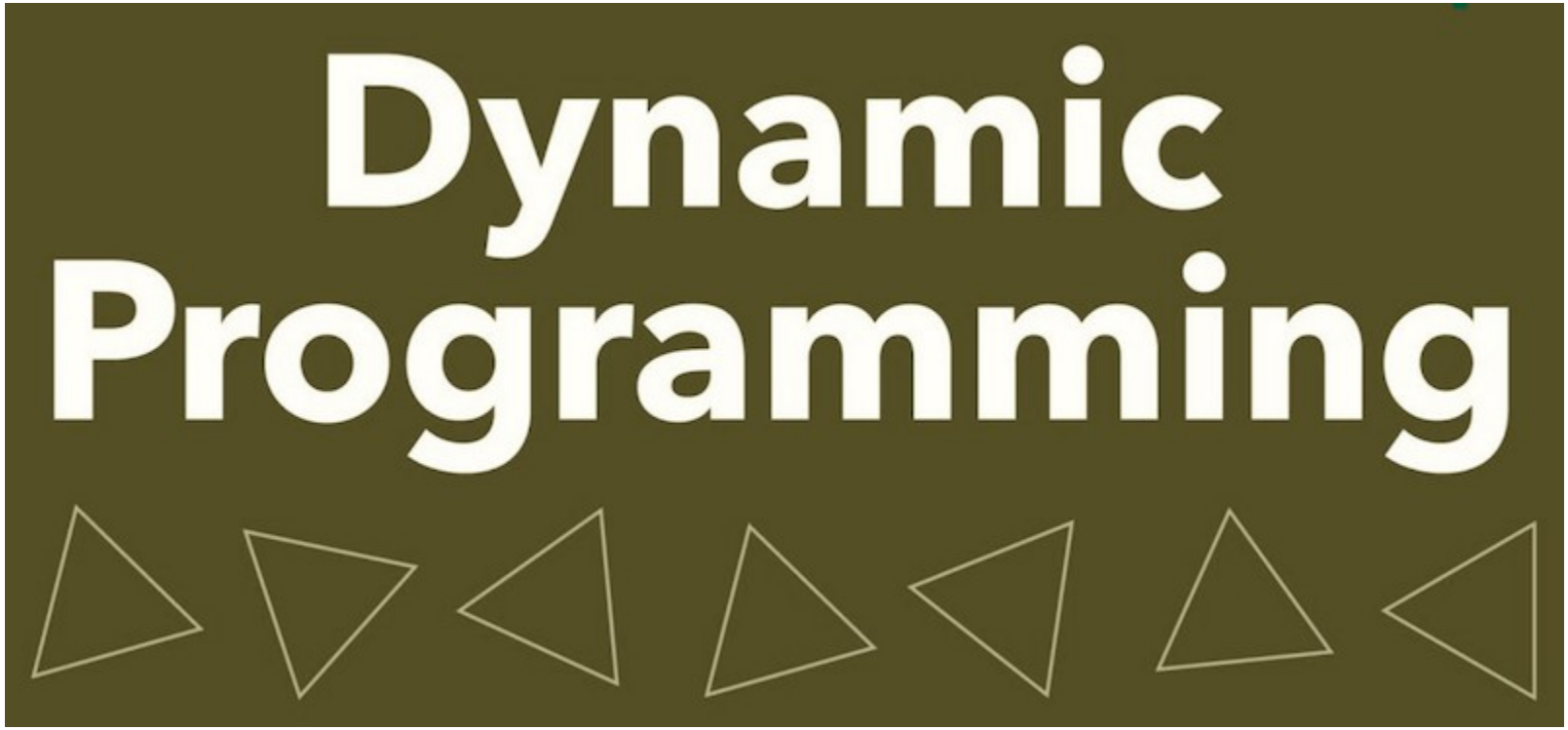
- Leetcode Patterns 2
- Depth-First Search (DFS) vs Breadth-First Search (BFS) – Techie Delight

Videos

- Breadth First Search Algorithm | Shortest Path | Graph Theory
- Depth First Search Algorithm | Graph Theory
- Breadth First Search grid shortest path | Graph Theory

Questions

- Flood Fill
- Leetcode - Binary Tree Preorder Traversal
- Number of Islands
- Walls and Gates
- Max Area of Island
- Number of Provinces
- 279. Perfect Squares
- Course Schedule
- C/C++ Program for Detect cycle in an undirected graph
- 127. Word Ladder
- 542. 01 Matrix
- Rotting Oranges
- 279. Perfect Squares
- 797. All Paths From Source to Target
- 1254. Number of Closed Islands

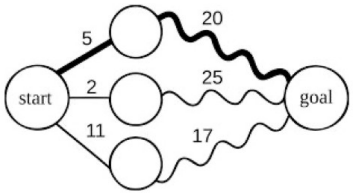

Dynamic
Programming

Dynamic Programming

Introduction

⋆ Dynamic programming is nothing but recursion + memoization. If someone tells you anything outside of this, share this resource with them. The only way to get good at dynamic

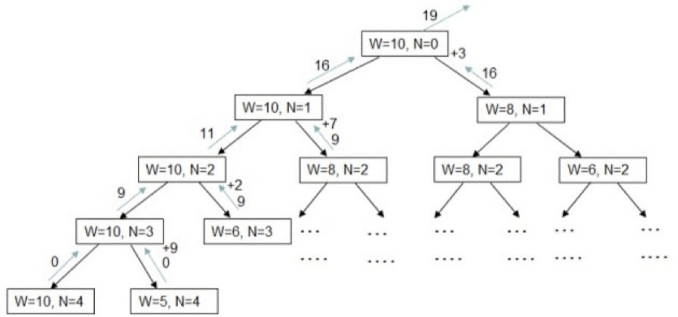
programming is to be good at recursion first. You definitely need to understand the magic of recursion and memoization before jumping to dynamic programming.
The day when you solve a new question alone, using the core concepts of dynamic programming -> you'll be much more confident after that.



So if you've skipped the recursion, backtracking, and memoization section -> go back and complete those first! If you've completed it, keep reading. You will only get better at dynamic programming (and problem solving in general) by solving more recursion (logical) problems.

Problem 1: 01 Knapsack

⋆ This is the core definition of dynamic programming. Understanding this problem is super important, so pay good attention. Every problem in general, and all DP questions have a CHOICE at every step.



Recursion tree for 0-1 Knapsack problem

Thinking about the arguments, a good recursive function would be passing in the weights, values, index, and the remaining weight? That way remaining_weight == 0 can be our base case. You can absolutely have other recursive functions with different arguments, it's about making things easier.

```
//include the ith item
int include = v[i] + knapsack(w, v, maxWeight - weights[i], i+1);
// don't include
int exclude =  knapsack(weights, values, maxWeight, i+1);
```

We think of the base case now. A straightforward one looks like maxWeight === 0, which is also the REMAINING weight as we're subtracting the weight every time we're iterating with the included item.
The second one and the most usual one is when you reach the end of the array, so index === weights.length. Can also be values.length as they're the same.
Here's the code for it:

```
knapsack(weights [], values [], maxWeight = 0, index = i, memo_set = set())
{
  if(i ==  weights.length || maxWeight == 0){
```

Dynamic Programming Playlist | Coding | Interview Questions |...

Aditya Verma

50 videos 8,160,685 views Last updated on Nov 1, 2020

Play all Shuffle

This playlist explains Dynamic Programming in a concise way. Explaining how to approach a Dynamic Programming problem and moreover how to identify it first.

1 unavailable video is hidden

1 

Dynamic Programming | Introduction
Aditya Verma • 1.2M views • 3 years ago

2 

2 Types of knapsack
Aditya Verma • 377K views • 3 years ago

3 

3 01 Knapsack Recursive
Aditya Verma • 477K views • 3 years ago

4 

4 01Knapsack Memoization
Aditya Verma • 389K views • 3 years ago

5 

5 01 Knapsack Top Down DP
Aditya Verma • 485K views • 3 years ago

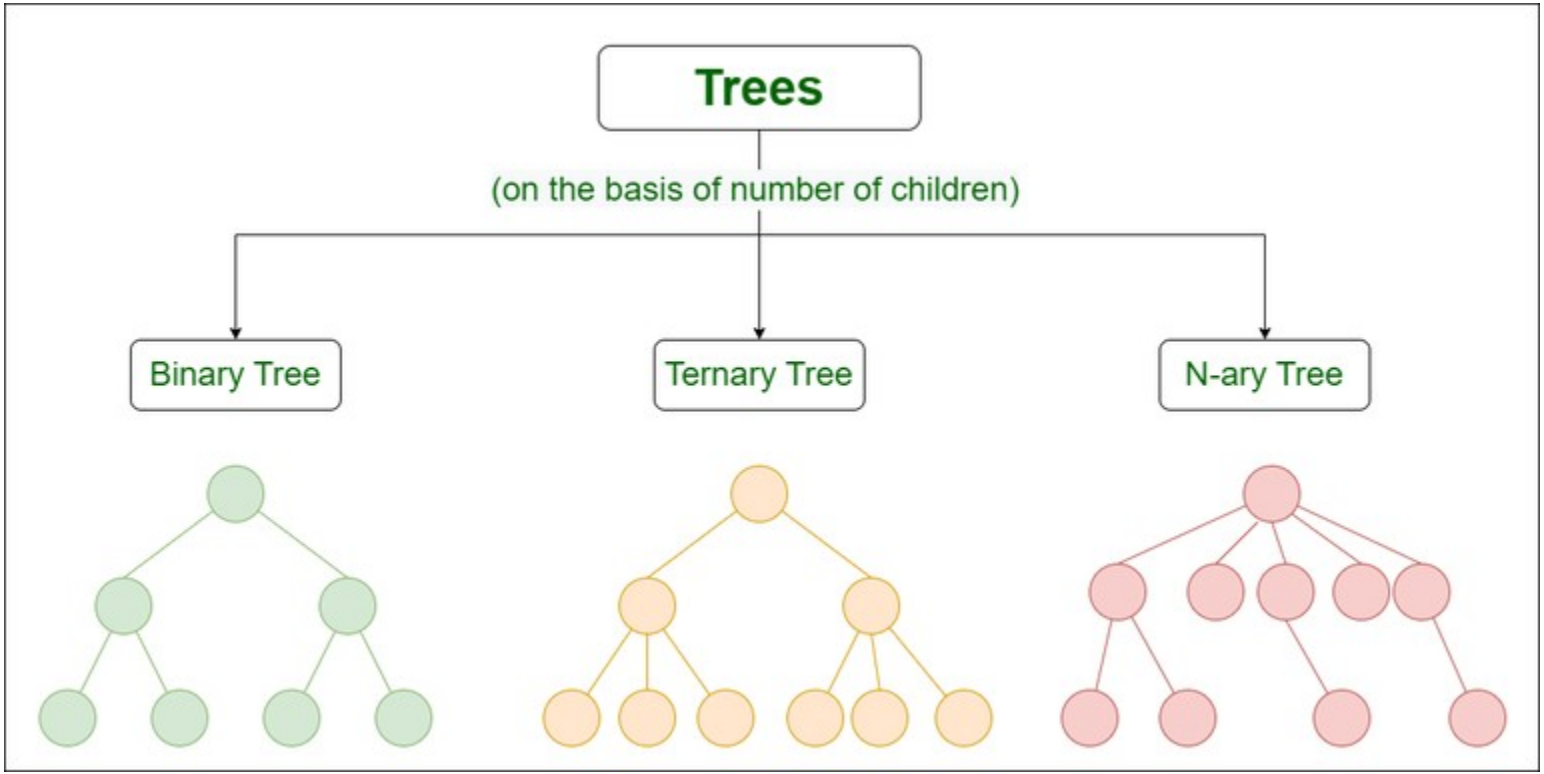
6 

6 Identification of Knapsack Problems and Introduction
Aditya Verma • 219K views • 3 years ago

7 Subset Sum Problem

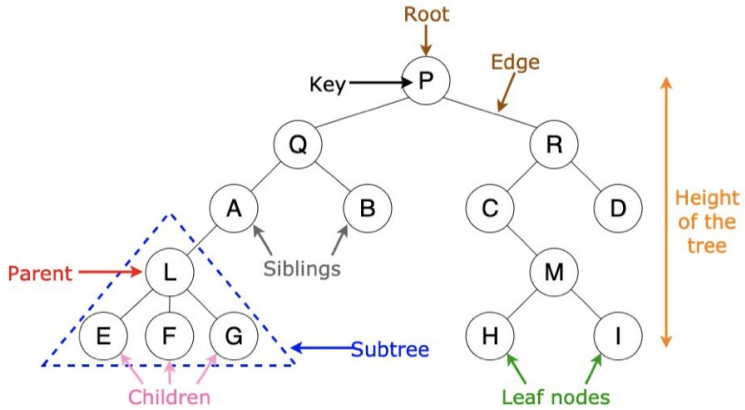
- https://www.youtube.com/playlist?list=PL_z_8CaSLPWekqhdCPmFohncHwz8TY2Go

- Best DP playlist
- For hindi - codebix


Trees
(on the basis of number of children)
Binary Tree
Ternary Tree
N-ary Tree

Introduction

⭐ I love trees, but actual ones - not these. Just kidding, I love all data structures. Let's discuss trees. They're tree-like structures (wow) where we can store different things, for different reasons, and then use them to our advantage. Here's a nice depiction of how the actually look:



Recursion is a great way to solve a lot of tree problems, but the iterative ones actually bring out the beauty of them. Making a stack and queue, adding and popping things from that, exploring children, and repeating this would definitely make sure you understand it completely. You should be seeing this visually in your head, when you do it iteratively.

Pattern: Traversals

⭐ There are 3 major ways to traverse a tree and some other weird ones: let's discuss them all. The most famous ones are pre, in, and post - order traversals. Remember, in traversals -> it's not the left or right node (but the subtree as a whole).

We start with the root, move until it's null or the stack is empty. We move to the left if we can, if not -> we pop, add the popped value and then move right.

```
List<Integer> res = new ArrayList<>();
if(root==null) return res;

Stack<TreeNode> stack = new Stack<>();
TreeNode curr = root;
while(curr!=null || !stack.isEmpty()){
    if(curr!=null){
        stack.push(curr);
        curr = curr.left;
    }else{
        curr = stack.pop();
        res.add(curr.val);
        curr = curr.right;
    }
}
return res;
```

Pre order traversal

⭐ We add the root, then the left subtree, and then the right subtree. It's a stack so things work in the opposite direction -> first in last out, so make sure to check that carefully.

```
Stack<Node> stack = new Stack();
stack.push(root);
result = [];

while (!stack.empty())
{
    Node curr = stack.pop();
    result.push(curr.data);
    // print node
```

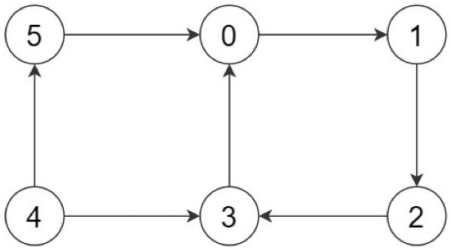
Graphs

Introduction

⋆ A lot of graph problems are covered by DFS, BFS, topo sort in general -> but we're going to do a general overview of everything related to graphs. There are other algorithms like Djikstra's, MST, and others - which are covered in the greedy algorithms section.

A lot of graph problems are synced with other types = dynamic programming, trees, DFS, BFS, topo sort, and much more. You can think of those topics sort of coming under the umbrella of graph theory sometimes.

Problem 1: Finding the root vertex

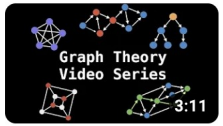


⋆ A human way of finding the root will be to look at 4 and say that there are no incoming edges at 4, so it's the root. Think of it in a tree like format, where the root is at the top and we have children below it.

How do we code this?

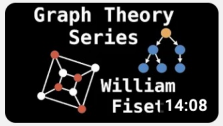
Graphs

- Searches
- Finding something
- Flights, comparison Qs
- Coloring, visited, etc
- Cyclic components
- Connected components

1 

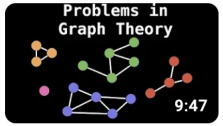
Graph Theory Algorithms

WilliamFiset • 173K views • 3 years ago

2 

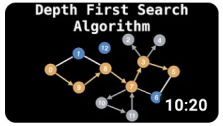
Graph Theory Introduction

WilliamFiset • 121K views • 5 years ago

3 

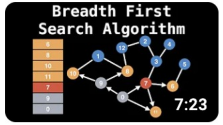
Overview of algorithms in Graph Theory

WilliamFiset • 72K views • 5 years ago

4 

Depth First Search Algorithm | Graph Theory

WilliamFiset • 367K views • 5 years ago

5 

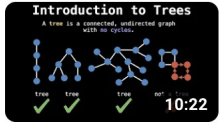
Breadth First Search Algorithm | Shortest Path | Graph Theory

WilliamFiset • 532K views • 5 years ago

6 

Breadth First Search grid shortest path | Graph Theory

WilliamFiset • 287K views • 5 years ago

7 

Introduction to tree algorithms | Graph Theory

WilliamFiset • 65K views • 3 years ago

- https://www.youtube.com/playlist?list=PLDV1Zeh2NRsDGO4--qE8yH72HFL1Km93P

- William fiset - amazing graph

ADDITIONAL TOPICS - IMPORTANT