Date:

"BIG O" (Important topic)

- This concept will make me a better developer.
- "Big O" will help us how well we can perform, basically it help us to optimize our code, and increases efficiency. (Asymptotic Notation)

Q: what is "Good Code"?

1) Readable → Clean, formatted code
2) Scalable → "BIG O"

For every task to be performed, we give instructions to the System/pc, in a way that it should be performed efficiently.

Example: Finding Nemo.

```
Const nemoArray = ['nemo', 'bingo'];
function findingNemo(nemoArr){
    for(let index = 0; index < nemoArr.length; index++){
        if(nemoArr[index] === "nemo"){
            Console.log("Nemo Found");
        }
    }
}
findingNemo(nemoArray);
```

Basically, these are the set of instructions that we have given to the System for finding Nemo.

Date:

Finding the performance / time taken by the pc to execute the respective code.

In Javascript, we use

performance.now()

- Basically Big O is a language that we use for taking, how long an algorithm takes to run.
- Question:

When we go bigger and bigger with our input, how much does it affect the function time / performance of the code. → this is what we used to say when we talk about Big O and Scalability of the code.

1- O(n) → Linear Time (most Common). → How much the run-time Increases?

2- O(1) → Constant Time → No matter how much the input is it is just Constant.

```
function funChallenge(input){
    let a = 10;          // O(1)
    a = 50+3;            // O(1)
    for(let i=0; i<input.length; i++){ // O(n)
        anotherFunction();   // O(n)
        let stranger = true; // O(n)
        a++;                 // O(n)
    }
    return a;            // O(1)
}
```

Means BIG O(3+4n).

Omega
Big O Nota
Theta Nota

Date:

- Omega Notation (Ω) gives the best case complexity.
- Big O Notation (O) gives the worst case complexity.
- Theta Notation (Θ) gives the average case complexity.

Rule Book for Big O calculations.

Rule 1: Worst Case (important).
Rule 2: Remove Constants
Rule 3: Different terms for Inputs (Important for interview)
Rule 4: Drop non Dominants

Common Interview Question :-

// log all pairs of array. (Complexity = $O(n^2)$)

because of the nested loops.

Another Example :-

```
function compressBoxesTwice( boxes, boxes2){
    boxes.forEach(function(boxes){ // O(a)
        console.log(boxes);
    });
    boxes2.forEach(function(boxes){ // O(b)
        console.log(boxes);
    });
}
```

As per the rule 3, these are 2 separate functions, So it will be $O(a+b)$, not $O(n+n)$

Date:

Rule #4: Remove non-Dominant Terms

for example:-

```
function printAllNumbersThenAllPairSums (numbers) {
    console.log('these are the numbers:'); // O(1)
    numbers.forEach (function (number) { // O(n)
        console.log (number); // O(n)
    });

    console.log('and these are their Sums:'); // O(1)
    numbers.forEach (function (firstNumber) { // O(n)
        numbers.forEach (function (secondNumber) { // O(n)
            console.log (firstNumber + O(n)
                secondNumber);
        });
    });
}
```

$= O(1) + O(n) + O(n) + O(1) + O(n) + O(n^2)$.

$= O(2) + O(3n) + O(n^2)$.

As per the Rule 4:

we will remove the non-dominant terms.

then it becomes.

$= O(n^2)$.

e:

(n!) : the most expensive one, the most Steapest.
: we are not going to encounter this, but if you see it, ~~there~~ than there is something gona wrong.

- Time Complexity of the built-in functions/lookups vary longugge to longuage.

For Example

"Sohaibisobedient".length ← In Javascript

The Space Complexity of this is O(1), and not the O(n

Gives us the idea to see how Scalable the Code is.

Date:

Summary Section:

- ✓ O(1): Constant - no loops.
- O(log N): Logarithmic - usually Searching algorithms have log(n), if they are Sorted (Bin Search).
- ✓ O(n): Linear - for loops, while loops
- O(2^N): Exponential - Recursive algorithms that solve a problem of size N.
- O(n log(n)): Log Linear - Sorting operations usually.
- ✗ O(n!): You are adding a loop for every element.
- ✓ O(n^2): Quadratic: Every element in a Collection needs to be compared to every other element. (Two nested loops)

Interview

Iterating through half a Collection is still O(n)

Two Separate Collections O(a * b)

2 separate inputs

What Causes time in a function?

- Operations (+, -, *, /)
- Comparisons (<, >, ==)
- Looping (for, while)
- Outside function Calls (function())

Rule Book:-

Rule 1: Always worst case

Rule 2: Remove Constants

Rule 3: Different inputs Should have different Variables. O(a+b)

A and B arrays nested would be O(a*b).

+, for Steps in order, * for nested Steps.

Rule 4: Drop Non-Dominant Terms.

e:

We should always write Scalable code, because we never when the inputs becomes Larger.
- This is why we use the concept of big O.

Good Code:

Data Structures + Algorithms = Programs.

Are the ways to Store Data

function/ways to write Data Structures.

Good Programmers; chooses good DS & A to write good Codes.

Remember:
1) Readable
2) Scalable — it should be, { Speed/time, Memory }

Pillars of Programming

- which is Supposed to be the best Code.
  1. Readable
  2. Memory (Space Complexity)
  3. Speed (Time Complexity)

Usually inverse of each other.

Sometimes either of them is more important.

We have to sacrifice for either thing.

Recursions

- what causes space complexity?
  1. Variables — heap
  2. Data Structures
  3. Function Calls — stacks
  4. Allocations

An Engineer (Good) must know, the runtime behavior, memory allocations.

Date:

Module 4: How to Solve Coding Problems.

- So what are companies looking for:
  1. Analytical Skills
     (related to your thought process)
     How can you think of a problem and analyze things, your ability of how → well you can think of solving a problem
  2. Coding Skills → Code well, organized, readable.
  3. Technical Skills → related background knowledge. Fundamentals
  4. Communication Skills

- We should know how to find the solutions
- We should know the "why" of doing things.

What we need for Coding Interviews:

most common Questions

Data Structures

1) Arrays
2) Linked lists
3) Trees
4) Hash Tables
5) Stacks
6) Queues
7) Graphs
8) Tries

Algorithms

1) Sorting
2) Dynamic Programming
3) Recursion
4) BFS + DFS (Searching)

All of these with the following:

1) Readable
2) Scalable → Memory (Space Complexity)
   → Speed (Time Complexity)

These are all that I should/must know, because 90% (even more) interviews are based on these Concepts

1) clarification
2) think out loud
3) talk before you write
4) Test your solution

Module 5 : Data Structures Introduction.

DataStructures is a collection of values.

- Each DataStructure is good and Specialized for their own things (specific things).

Example :

- Blockchain
- Bag packs
- Cabinets
- Box

- Two parts of data Structures are there to learn.
  1) How to Build one.
  2) How to use it. (Important).

- we can access any address from RAM much more faster from the CPU, than a Storage (SSD, HDD).
- Operations that can be performed on Data Structures.
  1) Insertion
  2) Deletion
  3) Searching
  4) Sorting
  5) Traversal
  6) Access

Date:

Module 6: Data Structures Arrays.

- Organizes the values in the sequential order. these are like a lists.
- Stored in contigious memory./Sequential Order
- Lookup / access — O(1)
  Push — O(1)
  Insert — O(n) } Linear time
  Delete — O(n) }

// reference type (Important Concept)

- Arrays are also an object.

reference

```
var object1 = { value: 10 };
var object2 = object1;
var object3 = { value: 10 };
```

- Strings related questions?

In an interview, we have to take the strings related questions, same as the arrays,

Question ie; reverse a string.

Question ie; merge two Sorted Arrays.

When to use Arrays?

1) Fast Lookups
2) Fast Push/pop.
3) Ordered

Good

1) Slow inserts
2) Slow deletes
3) Fixed Size*

* if using static array

Bad

Module 7: Data Structurs Hash Tables.

Hash Tables can also be called as "Hash maps", "maps", "unordered maps", "dictionaries", "objects", "Search Table", "Associative array".

Different Languages have different names for it and slight variations on the "hash tables".

1. JavaScript, objects are a type of hash tables.
2. Python, Dictionaries
3. Java, Maps
4. Ruby, Hashes

- Hash Tables are very important all across computer Science. Databases, caches, Associative arrays.
- Almost all the languages have built-in hash tables.

Hash Tables are great, when we want quick access to the certain values

- when to use hash tables?

Generally, no need to worry enough about it.

1) Fast lookups (Good collision resolution needed)
2) Fast inserts
3) Flexible keys

→ Good

1) unordered
2) Slow key iteration.

→ Bad.

They are useful when optimizing the Code.

encountered (nested for loops) $O(n^2)$

↓

optimized using hash tables

↓

becomes $O(n)$.

extremely useful interview Question!

Keep this technique in Mind.

Date:

2 arrays
check if they have
similar items?

for( )
  for( )

n²

Hashes.

{ } = firstArray

for (2nd Array) { }.

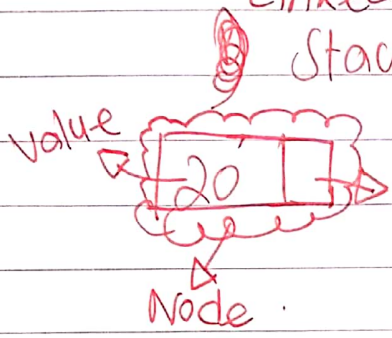
if { } console.log("yes");

Module 8: Linked Lists → Important interview.

There is always a trade off when it comes to the data structures.
we always have to compromis one thing over other.

If one DS is not available in that language, then we can always build one.

- JS is having a built-in garbage collector.
Linked can be used to implement.
Stacks / Queues.



Prepend	O(1)
Append	O(1)
Lookup	O(n)
insert	O(n)
Delete	O(n)

```
Class Node {
    constructor (value) {
        this.value = value;
        this.next = null;
    }
}
class LinkedList {
    constructor (value) {
        this.head = {
            this.value = v
            this.next = nu
        }
        this.tail = this.head;
        this.length++;
    }
}
```

Module 9: Stackes and Queues.

Stacks → LIFO
Queues → FIFO

✓ Fast Operations | ✗ Slow lookup
✓ Fast peek
✓ Ordered

Stacks		Queues	
lookup	O(n)	lookup	O(n)
Pop	O(1)	enqueue	O(1)
push	O(1)	dequeue	O(1)
peek	O(1)	peek	O(1)

add, push
pop, remove.

Browser web pages navigations, Redo, undo.

Printing Queue, ticket Booking

Usually we donot use lookup, we are only interested in Enqueue, Dequeue.

→ How does javascript works? Every task is performed by Javascript Engine.

what is program?
{ allocate memory
parse and execute }

Memory Leakage?
Something is declared in the memory but it is of no use, this is why we used to say "Global variables" are bad.

Call stack?
Javascript is a Single-threaded, So it has Single call Stack.
↳ only Single Statement runs at a time.

Synchronous programming.
- based on the call stack method, everything executes one by one.

```
Console.log(1);
Console.log(2);
Console.log(3);
```

example;
on a webpage, user has to do multiple tasks and if the first task is taking too long, then it is not as efficient as required.

what is Stackoverflow and how does it happens
it occurs when call stack is full and

this is where Asynchronous comes to rescue.

by using timeouts and breaks.

Slow but helpful.

this is not enough at all.

we need something, javascript runtime environment as well

part of the browsers.

they includes;
1) Web API's → 1) DOM
2) call back Queue. 2) AJAX (XMLHttp request)
3) Event loop. 3) Timeout (SetTimeOut).

it is already a part of Web Api.

example:-

```
Console.log('1');
setTimeout(() => {
    console.log('2');
}, 2000)
console.log('3');
```

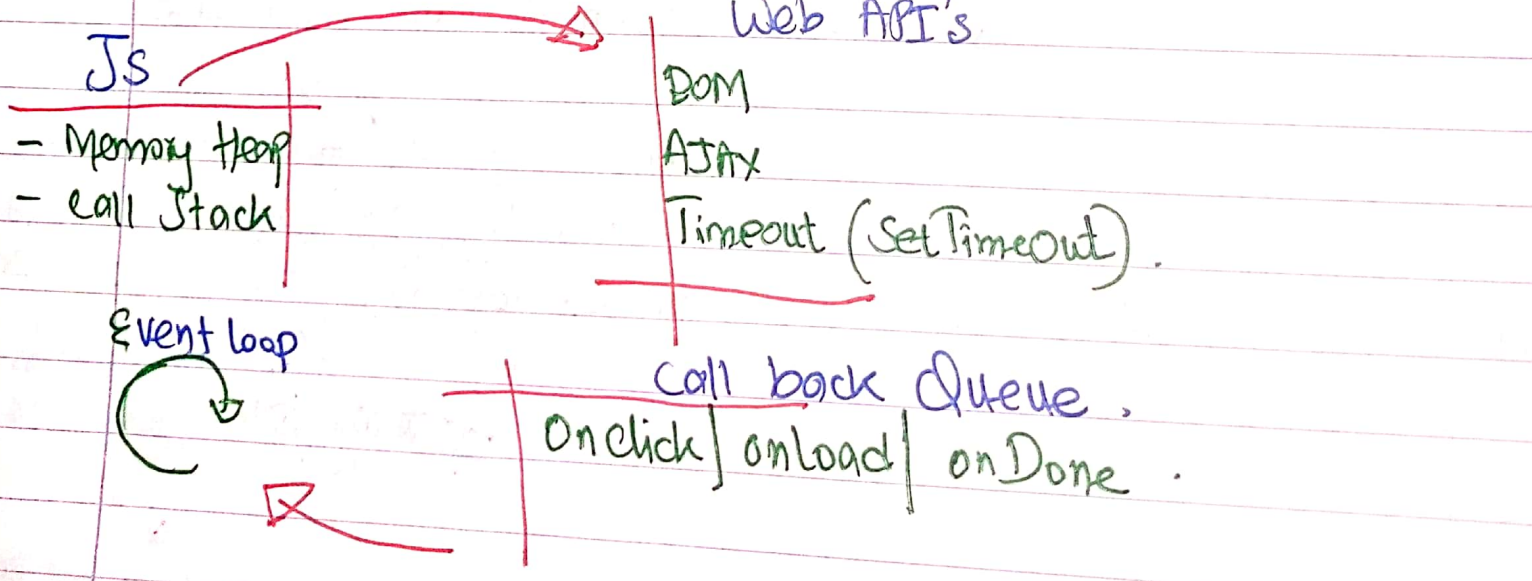
this will be sent to the runtime engine → webApi but will be executed after 2 seconds.

if we set it to "0" seconds, it will still go through the entire loop.

---

we want to get all the latest tweets, → worst idea because we have to wait for it to execute first and then other tasks.

Everytime when an event listner happens, it goes to the call Stack in the call back Queue.



Important ✱✱ Implement Queue's using Stacks.

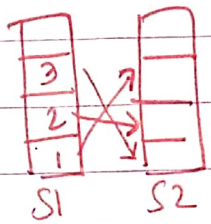
- Fast operations.
- Fast peek.
- Ordered

Slow lookup
↓
we donot use it.

FIFO
.. . .. Q
←
LIFO : ↓ S

peek, push, pop

push O(1).
pop O(n) → costly.



Implement Stacks using Queues.

FIFO

1	2	3	4	5

First all elements of Q1 in Q2.
push O(n) ← costly
pop O(1)

Date:

Module : 10 Trees.

arrays, LinkedList, Stacks, Queues.

↓

Linear.

Trees

↓

Non-linear. → are very important because we use this in our daily life.

- every child has only one parent.
- however one parent can have multiple.

↓

applications

→ Decision based games → for the best outcome.

→ facebook comments.

→ have nodes like LinkedList.

Binary Tree.

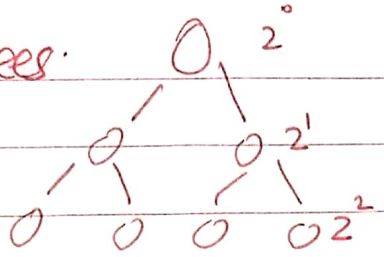
- each node can only have 0, 1, 2 nodes.
- each child can have only 1 parent.

Just like double linked list.

```
function binaryTreeNode(value) {
    this.value = value;
    this.right = null;
    this.left = null;
}.
```

Perfect Binary → very efficient.

Perfect Binary Trees.



very efficient.

1) double the nodes on every level.
2) total number of leaf nodes = sum of all upper + 1.

because we don't need to check every elem.

Lookup (O(log N)).
insert O(log N).
delete O(log N).

O(log n) → where we have possibilities.
only one has to be choosen.
↓
no need to check all the nodes.
↓
example; phone book.
↓
finding a name.

- example; google search engine.
↓
instead of O(n).
↓
it should be O(log n).

BST.

preserves relationships ← X not is hashtables
↓
example; computer folders.

- Conditions.
1) right node have larger value than parent.
2) Largest value = right most.
3) mininum value = left most.

~~Better than O(n)~~

~~No O(1) operations.~~

Date:

balanced vs unbalanced BST.

balanced → to have performance optimization.

unbalanced → why bad? → O(n) because of the biasness.

BST:

Better than O(n) → O(logn).
Ordered.
Flexible size

→ it should be balanced

→ Otherwise O(n).

no O(1) operation.

hash tables: lookup, delete → O(1).

We always have prepared libraries, so no need to remember the syntax.

While loop only

insert() → BST, use the rules on the previous page.
lookup()

remove()

↙ won't asked in the interview.

e: How to balance the BST.

↓

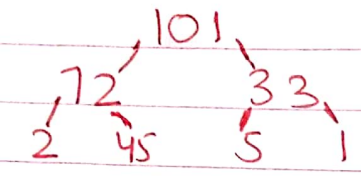
usually, in a production we do not have a balanced tree.

1) AVL tree.
2) Red and black tree.

Date:

Binary Heaps:

Lookup (O(n))
insert (O(log(n))
delete (O(logn))



1) parent node should be greater than left and right child. ⟶ max heap.
2) Left and Right child can be any value, unless greater than parent.

opposite of this ⟶ min heap.

Applications:
1) priority Queues.
2) Sorting Algorithms.
3) Data Storage.

heap

Add value in order from left to right,

Memory Heap != Heap Data Structure

these are not same.

heap of memory ↓ arbitrary data

priority queue

No need balanced bec

Priority Queues / Binary Heap.

No need to balance, they are already balanced because of the left to right insertion.

- this is not FIFO, now every element has a priority.

VIP/ or element having greater value will be served first.

Better than O(n) — Slow lookup.

Priority
Flexible Size
Fast insert

- only use PQ/BH for finding the max/min.
- not good for lookups.

Date:

Trie:

is a specialized tree used in searching most often with text.

and most cases it outperformance BST, hash tables.

- not binary tree.
- can have multiple child.

Big O ⇒ O(length of a word).

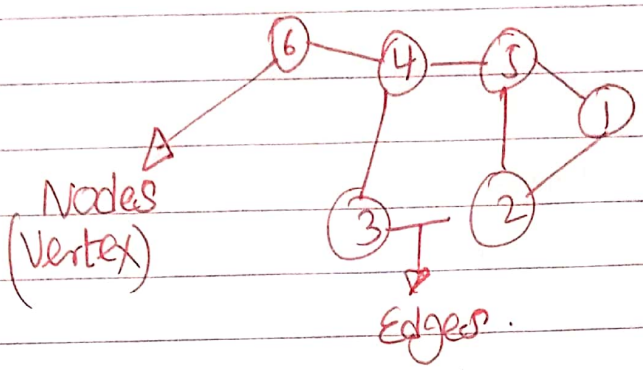
Space Complexity: Store at one location
↓
Use at multiple locations.

Graphs:-
most useful one
real life.

Linked list are a type of tree, are a type of graphs

Graphs :-

most useful and used when it comes to modelling real life.



Bellman_Ford.
Dijkstra.

- Reads.
- friends
- amazon for recommendations.

types of graphs :-

1) Directed → twitter is more directed. → used to represent traffic flow / one way
2) Undirected → high ways / Facebook friends.
3) weighted graph → google maps for finding the shortest optimal path.
4) Cyclic → when all the nodes are connected Circularly.
5) Acyclic → used in weighted graph.

Date:

```
Constructor(){
        this.numberOfNodes = 0;
        this.adjacentList = {};
}
addVertex(){
        this.adjacentList[node] = [];
        this.numberOfNodes++;
}
addEdge(node1, node2){
        // undirected
        this.adjacentList[node1].push(node2);
        this.adjacentList[node2].push(node1);
}
```

object.

Pros

- Relationships.

Cons.

- Scaling is hard

neo4j.com → builtin - graphs, for using in production.

blockchain is built using the DS.

ALGORITHMS.

- Sorting - Arrays / Trees.
- Dynamic Programming. - Hash tables.
- BFS + DFS (Search) Graph / Trees
- Recursion Trees.

Data Structures + Algorithms = Programs.

Class {} + functions() = programs.

named location that can be used to store and organize data.

Collection of steps to solve a particular problem

allow us to write efficient and optimized Computer programs.

→ BigO and Scalability are important. As company grows larger and larger

Date:

Recursion ← Common Topic for interview.

[redacted].

→ function refering to itself unless a certain criteria is achieved.

- finding files within folders and subfolders recursively.

: ls -R

Searching Algorithms.
Sorting Algorithms. ← Recursion is used everywhere.

Stackoverflow also comes under this when a recursion got limitless.

Maximum call stack size exceeded.

biggest problem with recursion.

it costs the memory.

→ should have some base case to stop the recursion.

we should take care of stackOverflow.

Anatomy of Recursion :-

we can use debuggers; to see the process step by step; in Chrome Dev tools.

```
let counter = 0;
function recursiveCalls(){
    if(counter = 5){
        return "Done!";
    }
    counter++;
    return recursiveCalls();
}
recursiveCalls();
```

Rules :-

1) Identify the Base case
2) Identify the recursive case.
3) Get closer and closer and return when needed. Usually have 2 returns.

- Factorial → examples.
- Fibonacci → creates a tree. $O(2^n)$.
  important.

Fibonacci series with recursive approach is more readable but is not as ideal approach as compared to the iterative approach.

RA = $O(2^n)$.
IA = $O(n)$.

Date:

Why recursion over iterative approach:

- anything that can be implement by recursion can also be done iteratively (loop).

- [illegible]

Pros	Cons
• DRY	• Large Stack
• Readability	

- will use recursion when we are not sure how many loops to go through / how deep they are.

- tree DS traversal ← useful.

Tail call optimization

↓

recursion to be called without increasing the stack. / So that they can be memory efficient.

When to use Recursion?

- BFS and DFS (Searching).
- Sometime in Sorting.

- Rules:-

Everytime you are using a tree or converting Something into a tree, Consider recursion.

1) Divide into a number of Sub-problems. that are smaller instances of the same problem.
2) Each instance of the subproblem is identical in nature.
3) The solution of each subproblem can be combined to solve the problem at hand.

- Divide and Conquer using Recursion.

Date:

Sorting Algorithms.

- very important for interviews.
- Why do companies care about sorting?

↓

when the input becomes larger and larger.

- On small operations, we really does care about it because we have fast computers for that; but company have to pay castly if it the input becomes so large/expensive.

↓

example; - Google web page sorting.
- amazon is all about sorting products based on recommendations.
- netflix
- microsoft.

↓

if having random data it will be really really hard to get exact data.

- cannot use built-in sort methods, so companies have to use the custom built sorting Algorithms.

1- Bubble sort
2- Insertion sort
3- Selection sort
4- Merge sort
5- Quick sort

Searching and sorting are 2 biggest kind of problems in Compute Science.

1. Bubble Sort
2. insertion Sort
3. Selection Sort
4. Merge Sort
5. Quick Sort

elementary

6) Radix Sort
7) Heap Sort
8) Counting Sort

different in different engines
Quick, insertion, merge, Combination.

Issue with sort() in JS.

- we should not always rely on the built-in functions.
- we should read the documentation of the respective language, what we want to use.

non comparison sort.

1) Bubble sort.
2) insertion sort.
3) Selection sort.

elementary Sorts → Should comes first in mind when ever the a time to sort

4) Merge sort.
5) Quick sort.

Complex but more efficient than elementary sort.

6) Heap sort.
7) Counting sort.
8) Radix Sort

non- comparison sort.

Date:

1. Bubble Sort :- (most simplest but not so efficient)

6 5 3 1 8 7 2 4

5 6 3 1 8 7 2 4

5 3 6 1 8 7 2 4

5 3 1 6 8 7 2 4

time complexity = $O(n^2)$.
Space complexity = $O(1)$.

2. Selection Sort.

xs - 8 1 red
s - 5 1 blue
s - 2
3 - 6
- 9
6 - 8 - 5
s - 1
4
s - 8
7

time complexity = $O(n^2)$
Space complexity = $O(1)$.

- Consider 1st item, the smallest one.
- is the iterator to find the smallest one.

3. Insertion Sort.

6 5 3 1 8 7 2 4

① 5 6 3 1 8 7 2 4

② 3 5 6 1 8 7 2 4

④ - ③ 1 3 5 6 8 7 2 4

⑤ 1 3 5 6 7 8 2 4

Merge

Efficient

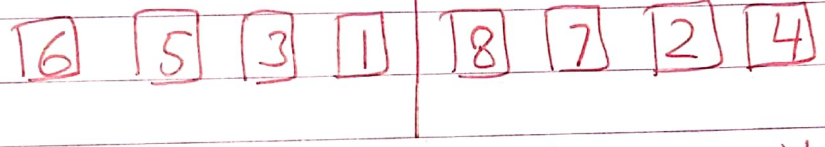
Merge Sort → time complexity. O(n log n).

Last big(o).

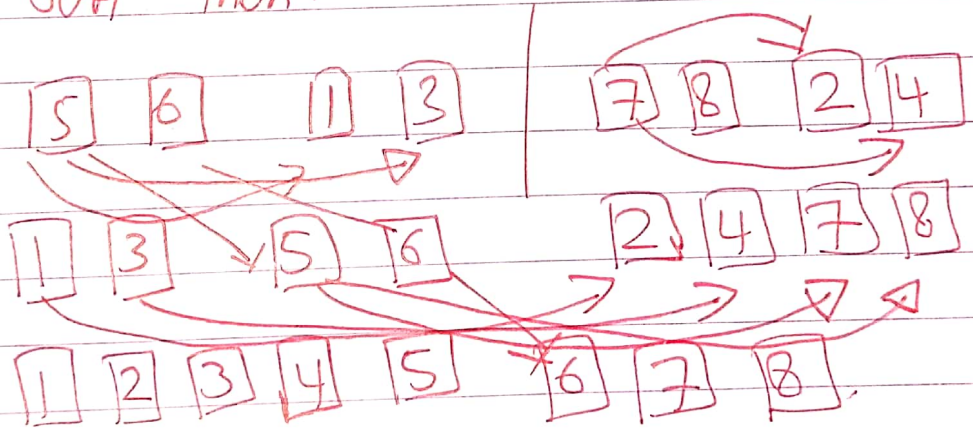
much better than Previous 3 Sorts.

1) bubble
2) insertion
3) Selection

merge sort
Quick sort → both uses the divide and conquer rule.



Divide the list into half until we have only 1 item left and then recursively Sort then.



Merge Sort is a great algorithm to use.

EASE

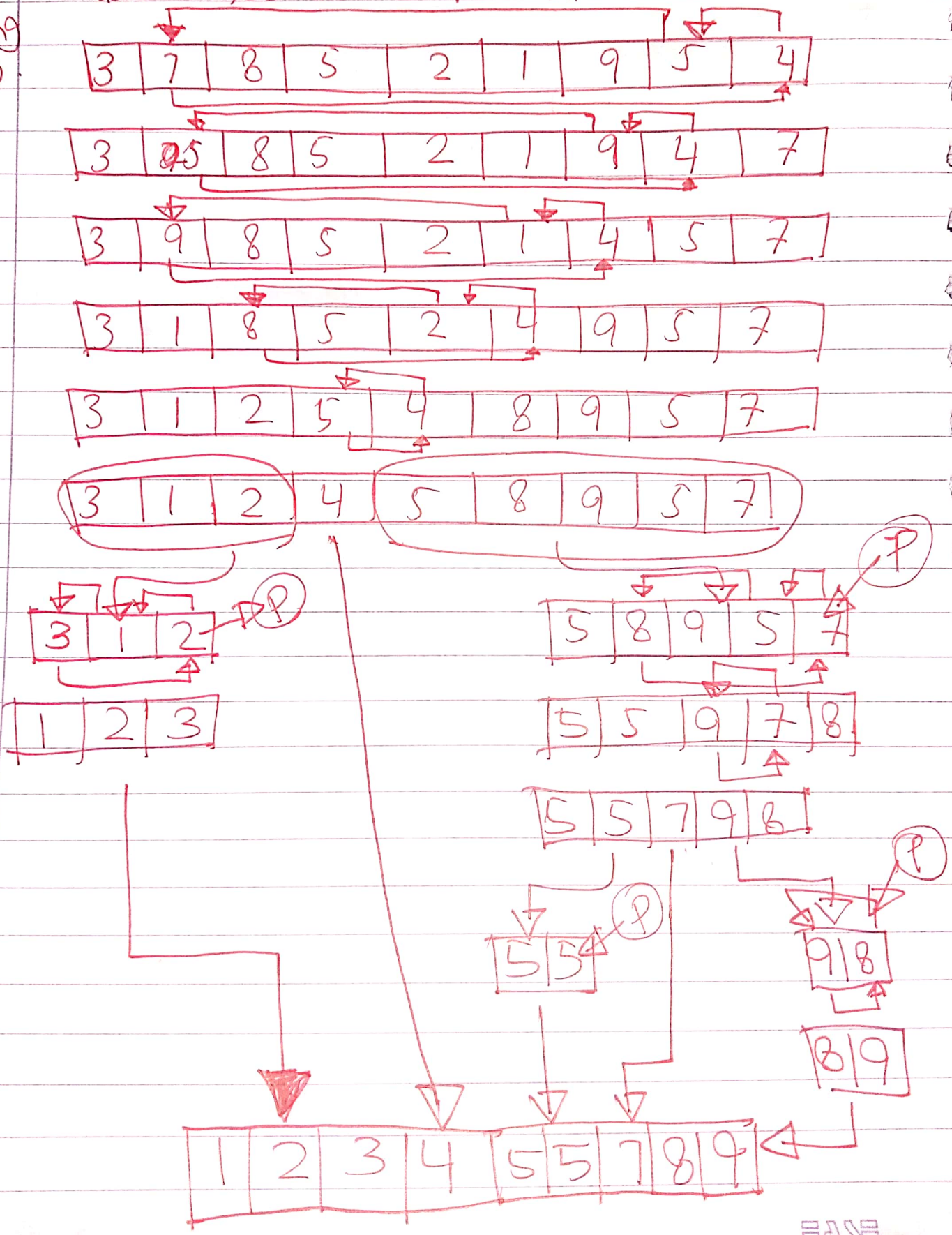
Pretty fast TC = $O(n^2)$

Date:

(5) Quick Sort → SC = $O(\log n)$.

best sorting Algo.

uses divide and conquer rule as well.



which sort is best ? if having user data of 100 M.

Insertion → for very small data.
bubble → not as efficient / not useful.
Selection → not as efficient / not useful.

Merge → is very efficient if we donot have problem with space complexity. O(n).

QuickSort → is the best of all, most popular. the algorithms, having good TC & SC. but if the pivot point is not pickes properly, than we are at a major risk because of the worst case. O(n²)

better SC than merge.

most popular.

than we will have a slow sorting.

heapSort → can be used if TC of Quick Sort is bothering

O(n log n)

by mathematically it is the best one.

Date:

Radix Sort + Counting Sort

↓

non-comparison Sort.

↓

Only work with numbers

↓

must have Limited range.

We can always implement the algorithms using the pre built Libraries / frameworks.

orted data is always more efficient.

Searching Algorithms.
↳ used every day.
↳ Searching is a big part of our Lives.

1. Linear Search
2. Binary Search
3. Depth Breath Search (DFS)
4. Breadth first Search (BFS).

- Linear Search . O(n).
  numbers, nodes, anything.
  → Simple traversal.
  TC = O(n).
  includes.
  indexOf.

- is there a better way to find a number?
  ↓
  Sorted.

- Binary Search. → Log(n)
  ↳ we know that List is sorted.

  1 ~~4~~ ~~6~~ 9 12 34 45 (34)

  ~~9~~ ~~12~~ 34 45

  found.

9 4 1 6 20 15 170.

Date:

Graph + tree traversals :-

when we have to iterate over every node then in both cases: graph, tree.
we have 2 choices.

1) BFS. O(n).
2) DFS.

BFS.

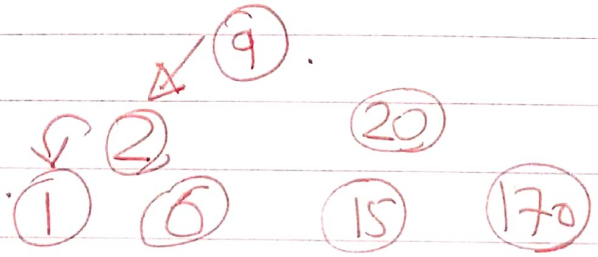
In BFS, we move left to right level by level to iterate over every node; until we find the given node / end of tree.
→ Starting from the root node.

→ we have to keep track of every visited node.

DFS. → keeping track is not required.

→ Starting from root node.

by Default → pre order.



9, 2, 1, 6, 20, 15, 170.

Because of the backtracking nature, we have to use recursion, in DFS.

ate: O(n).

BFS vs DFS.

When should you use one over other?

BFS → Closest friends on FB.

- Shortest path
- Closer nodes
- Memory

if we have additional information like which level is on the number then use BFS.

- for finding the Shortest path.

DFS:

Less Memory — Can get slow

Does path exist

- can get worse if the depth increases.
- Should not use for path finding.

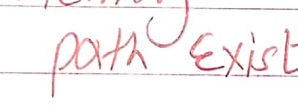
- this is how facebook friends recommendation works.
- how google maps works.

In DFS, we have 3 ways that can be implemented.

used for sorting. ① In order. 33, 101, 105

if we want to create a tree again. ② Pre order. 101, 33, 105

③ post order. 33, 105, 101



Date:

Trees are a type of graphs.

- Dijkstra
- Bellman-Ford

} for Shortest path.

We have BFS for Shortest path, then why these 2?

- because BFS donot have weights, it just gives the same weight to every path.
- Bellman works well when having -ve weights.
- Otherwise for +ve weights Dijkstra is the best.

ate:

Dynamic Programming : Divide & Conquer + memoization

it is an optimization technique.

Do you have something you can cache?

↓

Dynamic Programming.

Memoization. ~~~ Caching.

Just a way to speed up programs.

, having a back pack for faster process.

example:

Boy going to school, he wants pencil, so instead of going back to home totally, he just get it from the pencil box he is having.

```
let cache = {}
function memoizationAddTo80(n) {
    if (n in cache) {
        return cache[n];
    } else {
        console.log('long time');
        cache[n] = n + 80;
        return cache[n];
    }
}
```

Hashing - so fast!

2nd time - onwards.

1st time

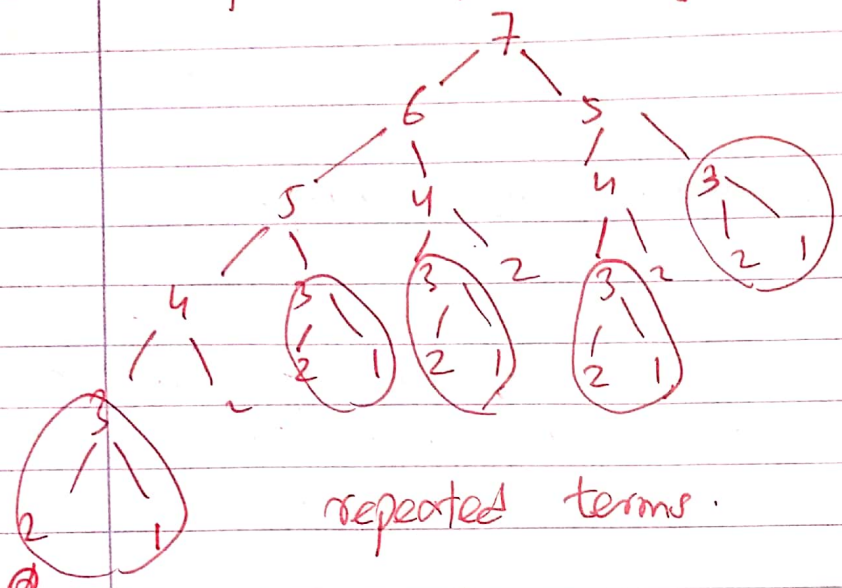
if parameter change, then have to run again.

3. here we can also use the colosure property.

```
return function(n) {
```

Date:

For example;
In fibonacci series, we have a repeated patterns for the factorials.



repeated terms.

- So we can use here the memoization function.

rules:-

1) Can be divided into sub problems
2) Recursive solution
3) Are there repetitive sub-problems?
4) Memoize subproblems.
5) Demand a raise from your bass.