Week 10

Graphs and Recursion

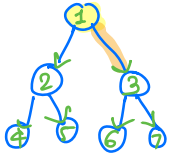
what's my motivation ⇒ Relationship between nodes

Data Structures so far..

linear

relationship bet^n nodes one stored after other

non-linear



relation between 2 nodes ✓

parent - child

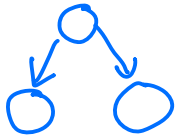
BST ⇒ set of nodes/vertices
set of edges (directed)

n = 7
edges ?
edges = 6
(n-1)

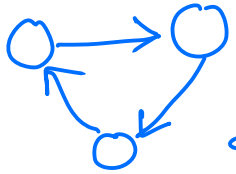
Restriction on BST

Restrictions BST

1. n nodes
   n-1 edges
2. directed edges
3. no cycles
4. less relationship



BST



Graph

directions ✓

All BSTs are Graph but not all graphs are BST

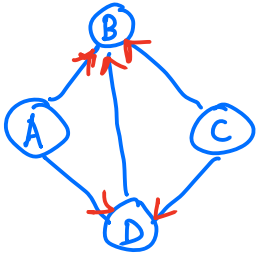
Graph ⇒ useful in relationship establishing

A graph G is an ordered pair of set V of vertices and a set E of edges

G = (V, E)

$(A, B) \neq (B, A)$ ⇐ ordered pair

$\{a, b\} = \{b, a\}$



$V = \{A, B, C, D\}$

$E = \{A, B\}, \{A, D\}, \{D, B\}, \{C, B\}, \{C, D\}$

OR

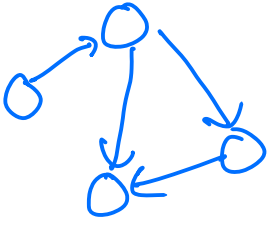
$\{B, A\}$

ordered pair ⇒ If directed, (A, B), (A, D) (D, B) (C, B) (C, D)

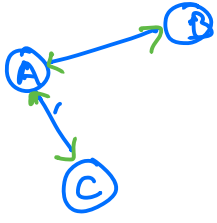
Directed and undirected graph

A graph G is an ordered pair of set V of vertices and a set E of edges

G = (V, E)



you can think of it as bidirectional graph

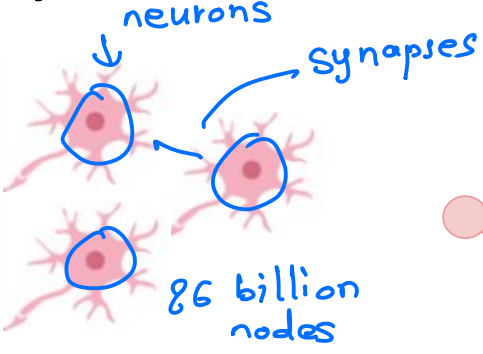


you can have mix of both

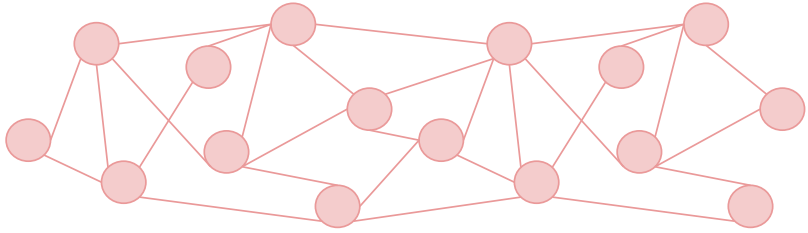
but in this class, we consider just one

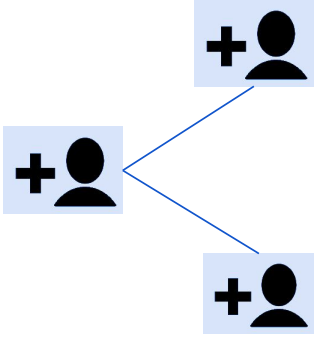
Graph Theory

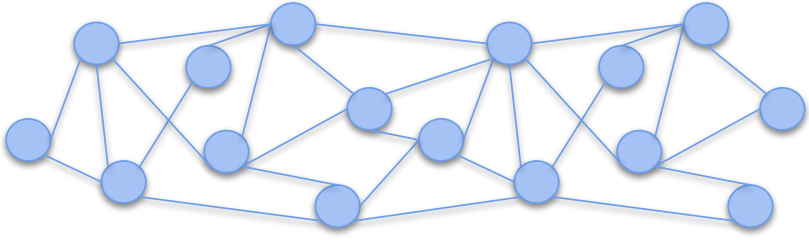
(Application)



Graph, bidirectional

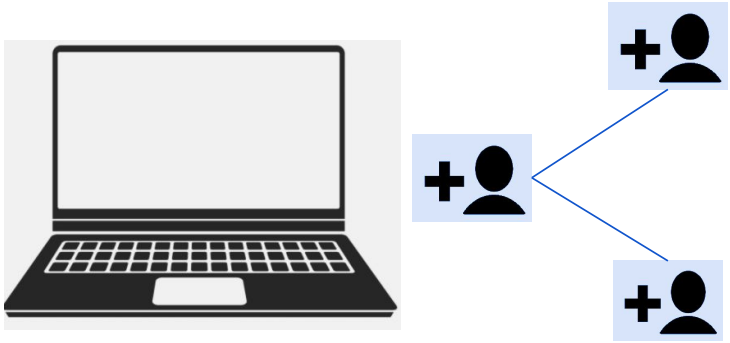


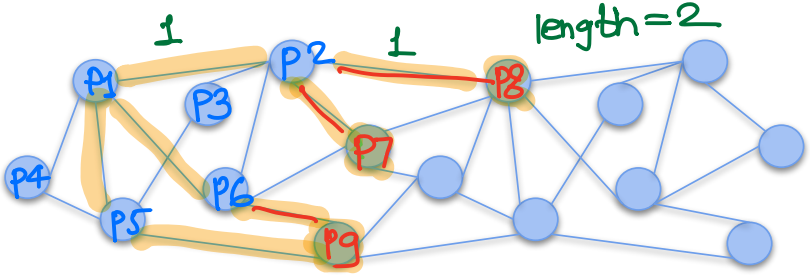




Graph Example - Social network

P1 → 5 friends





Suggests more friends for P1

⇒ Using graph traversal and find a path of length 2

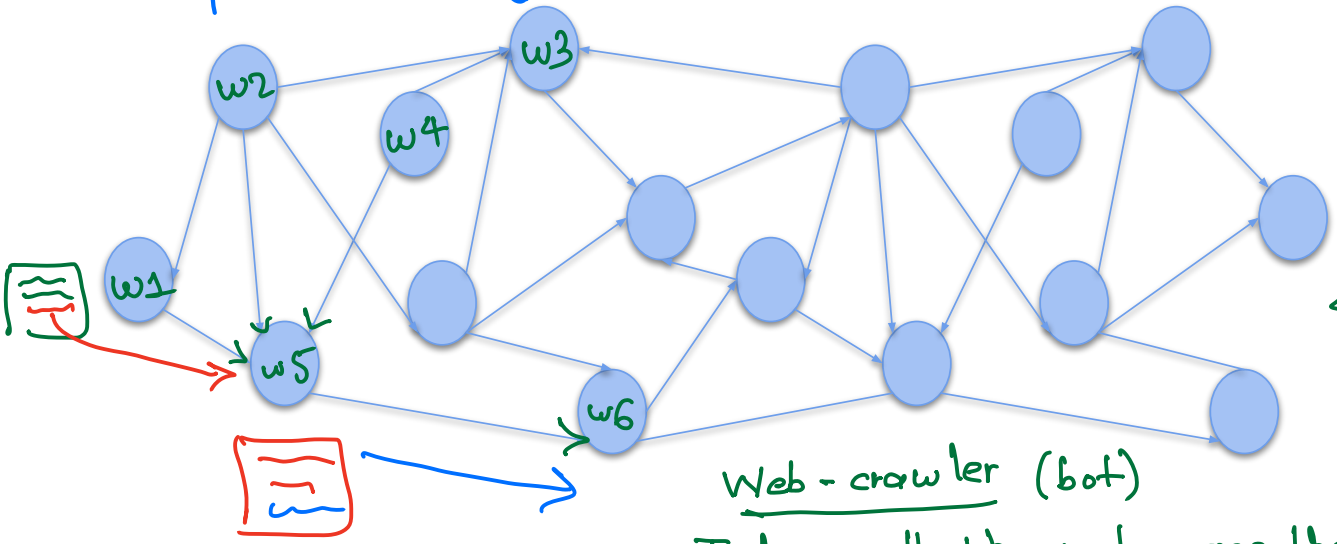
no direction

P1 friend with P2, P2 also friend with P1

Graph example: World wide web

⇒ collection of web pages
web pages have links to other web pages

w_3 and w_5 seem to be important

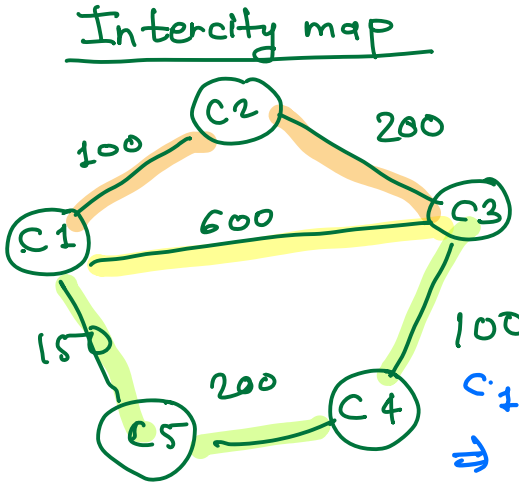


⇐ directed graph

Web-crawler (bot)

Indexes all the web pages that forms this n/w

Weighted vs Unweighted Graphs



best route with least distance

100

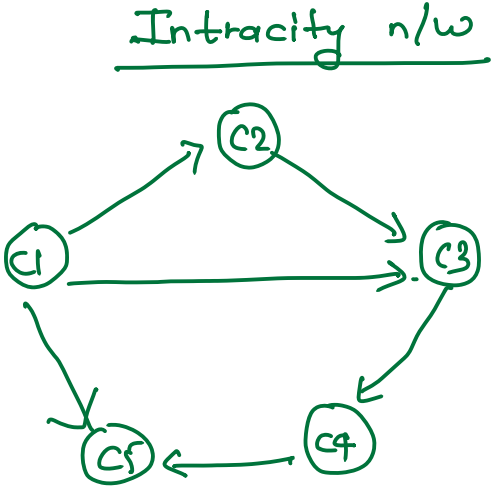
c_1 to c_3

⇒ 3 ways

⇓

choose least distance

undirected ∵ highways are undirected



directed (one-ways)

Another Graph Example

(maps all the words and establishes relationships)

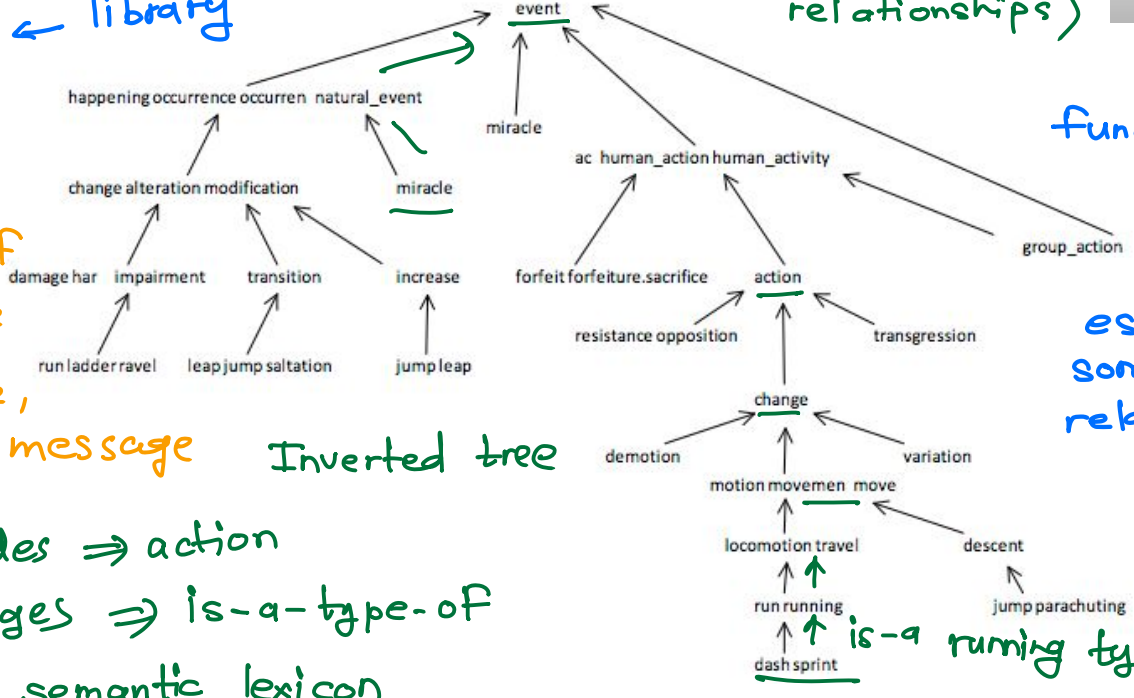
text sentiment analysis

textBlob ← library

⇓

overall polarity of a sentence

+ve, -ve, neutral message



Inverted tree

fundamental use of graph is to establish some relationship

nodes ⇒ action

edges ⇒ is-a-type-of

used in semantic lexicon

is-a running type

Each edge represents 'is-a-type-of' relationship

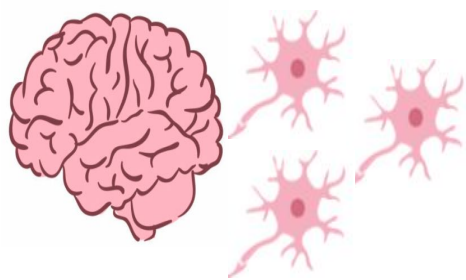
Graph Theory to model Pain processing

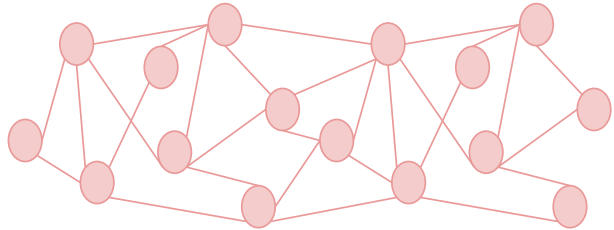
https://link.springer.com/article/10.1140/epjb/s10051-021-00046-6

maps pain activity in your brain

Neurons are nodes and synapses are edges

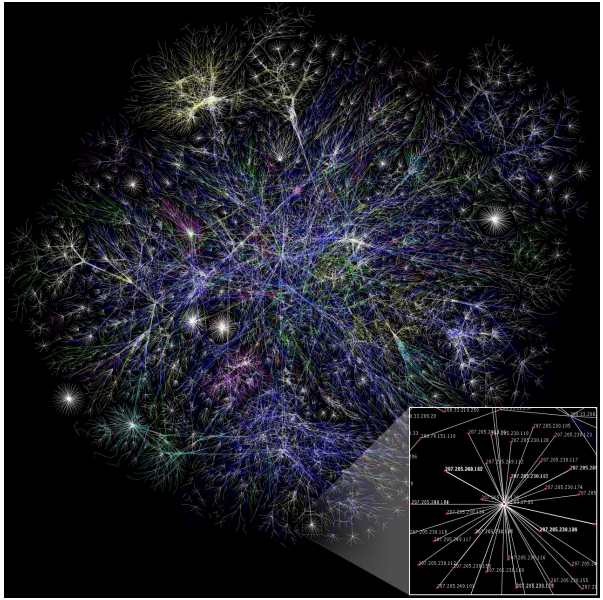
creates biologically plausible network to emulate brain activity during pain processes





(directed) + edge
weighted
Google → hop → 2 hop
hop
nodes ⇐ IP address
map of internet ⇒ map of IP addresses

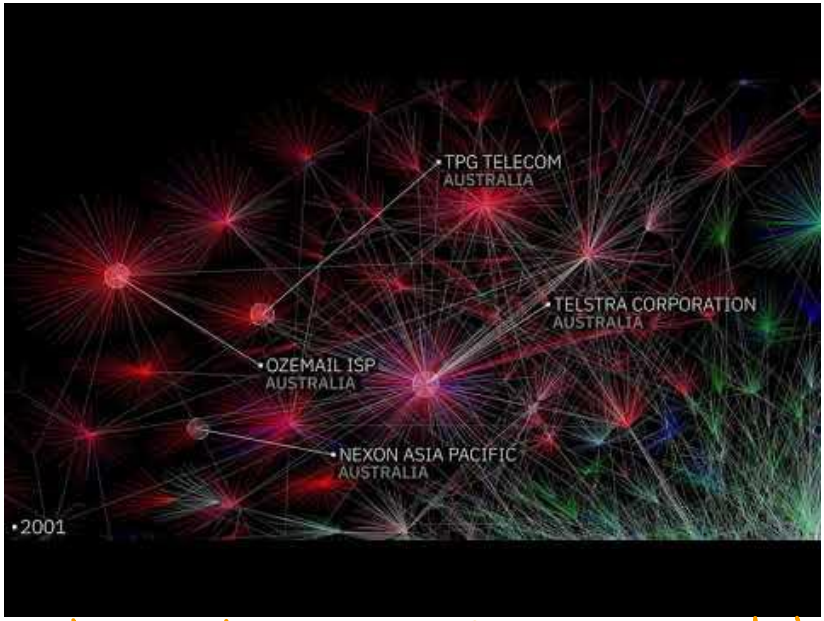
Graph of Internet



Internet map 2005
Image taken from Wikipedia:
Creative Commons Attribution 2.5

(30% of map)

North america ⇒ blue



Official website: https://www.opte.org/the-internet
Credit: Barrett Lyon

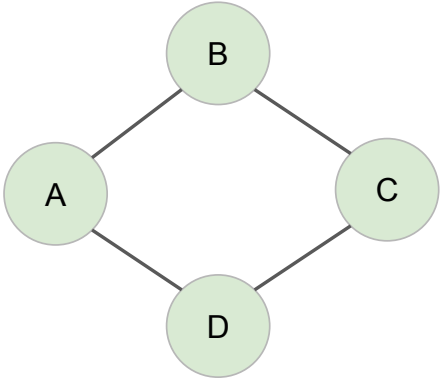
opte.org

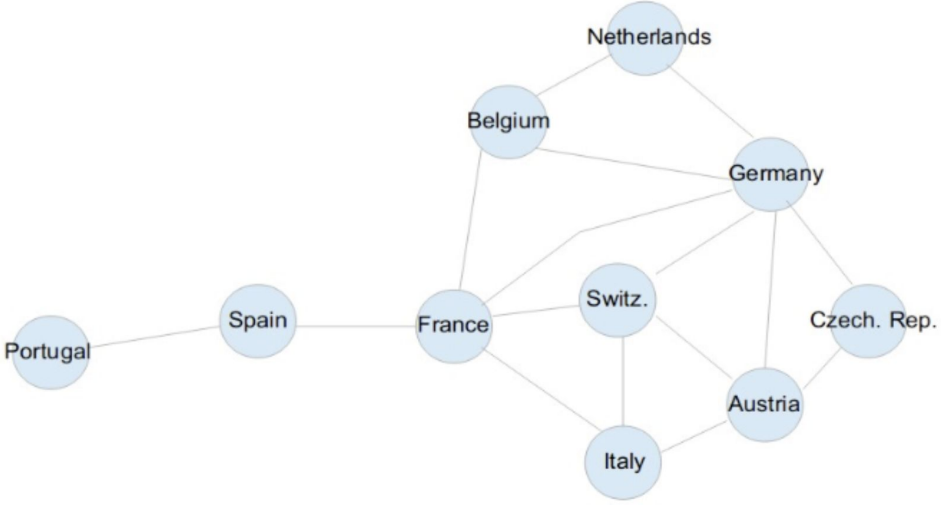
Some source code is open

Definition of a Graph

Ordered pair of set of **Nodes** (or vertices) and a set of **Edges**

G = (V, E)





Simple Graphs

No self loops

No multi-edge

Simple Graphs

self loops

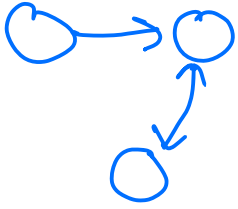
self loop

Any example?

recursion ✓

✓ web page

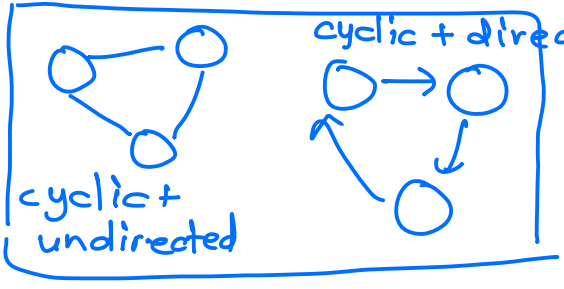
undirected + directed



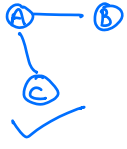
multi-edge

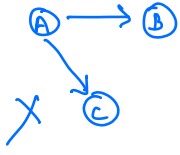
example?

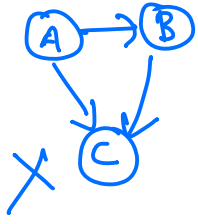
flight example



Connectedness of a graph

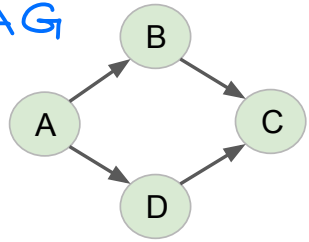
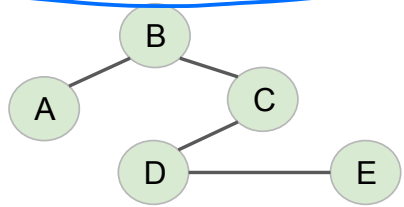
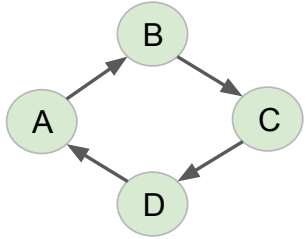
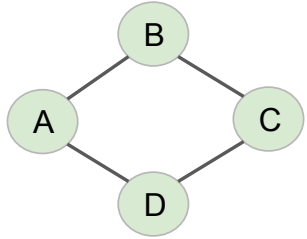




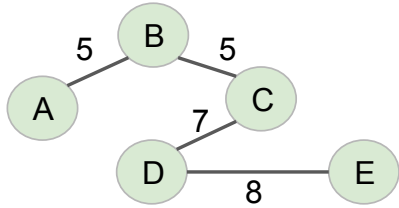


Types of Graphs

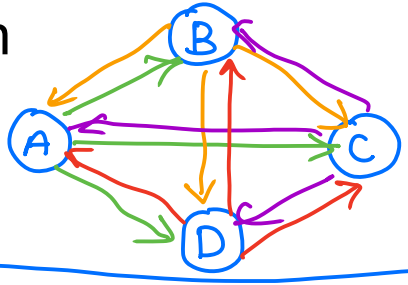
dist bet^n cities ↓ internet hops

	Directed	Undirected
Acyclic	DAG 	
Cyclic		

Weighted Graphs



Properties of graph



time complexity

Number of edges

Directed graphs ⇒ $|E| \Rightarrow$ max. no. of edges??

$|E| = 12$

$4 * 3 = 12$

$|V| = n$

$|E| \leq n * (n-1)$

Undirected graphs? $|E| \leq \frac{n(n-1)}{2}$

Dense vs Sparse graph

depending on no. of edges (subjective)

dense ⇒ many edges ⇒ Adjacency matrix

sparse ⇒ fewer edges ⇒ Adjacency list

No. of edges grow quadratically

$|V| = 10 \quad |E| \leq 90$

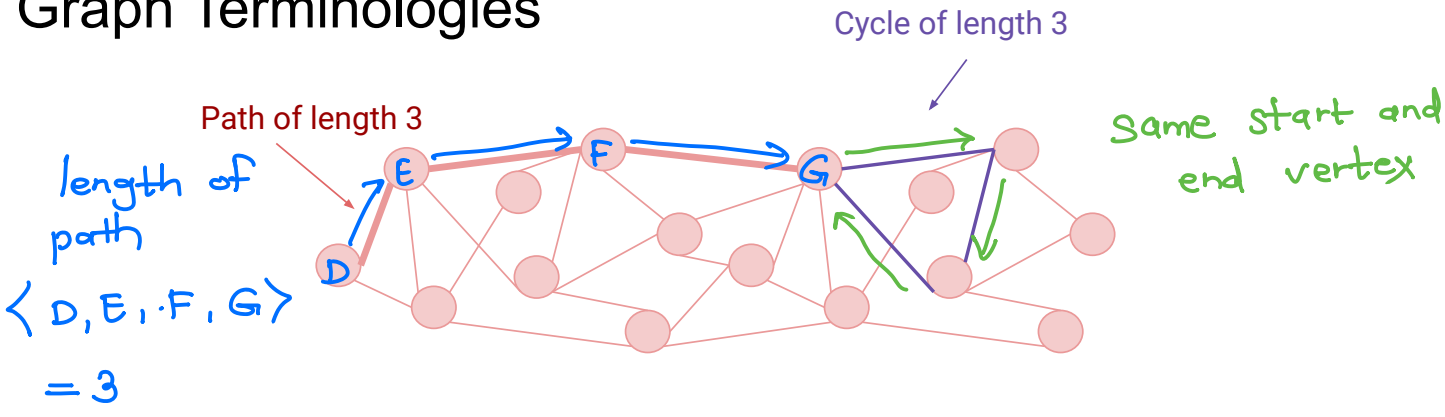
$|V| = 100 \quad |E| \leq 9900$

Aim is NOT to have complexity in terms of $|E|$

Social networking ⇒ sparse

dictates data structure, to store

Graph Terminologies

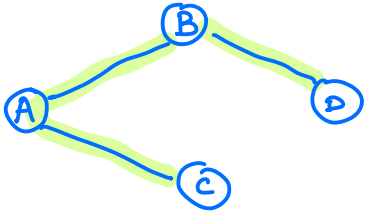


- Graph ***G = (V, E)***:
  - A set of **Nodes** (or vertices) and a set of **Edges.**
  - Vertices with an edge are **adjacent.**
- A **path** is a sequence of vertices connected by edges.
- A **cycle** is a path whose first and last vertices are the same.

Properties of graph

Connected graphs

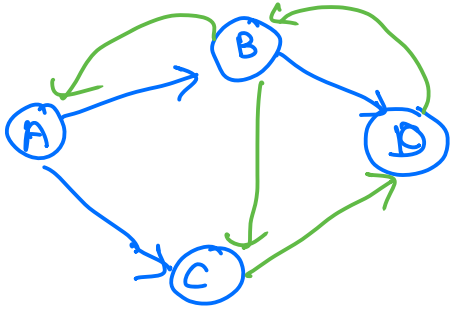
undirected



associated with undirected graphs

Strongly connected graph

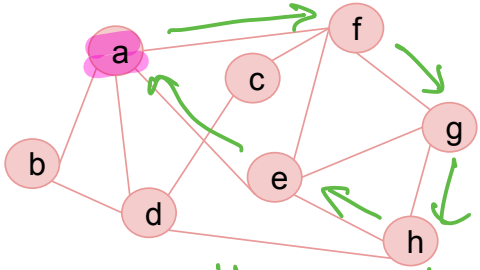
associated with directed graphs



(not strongly connected)

Graph Terminologies

Easy to answer questions by looking at graph. But to formalize theory and write algo. is difficult

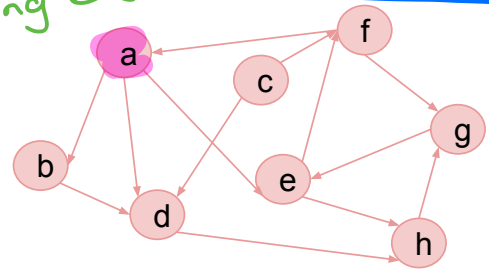


all the vertices that can be reached with path length = 1

- Neighbours $a = b, d, e, f$
- Neighbourhood (set of vertices) $a = \{b, d, e, f\}$
- Degree no. of nodes in neighbourhood $= 4$
- Path $\langle a, f, g, h \rangle$
- Cycles $\langle a, f, g, h, e, a \rangle$

sequence of consequtive nodes that can be reached

outgoing edges

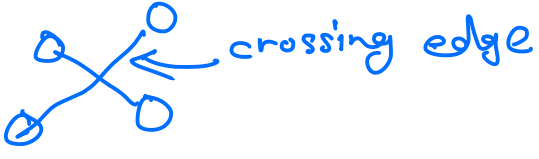


- Neighbours:
  - Out-neighbour : $a = b, d, e$
  - In-neighbour : $a = f$
- Neighborhood (Out and In Neighbourhood):
- Degree
  - Out-degree $= 3$
  - In-degree $= 1$
- Path $\langle a, e, f, g \rangle$
- Cycles $\langle a, e, f, a \rangle$

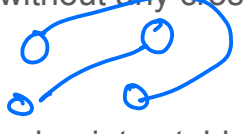
Important to answer graph related questions

Some common Graph-processing Problems?

- Path - Is there a path between nodes a and b? can be multiple
- Shortest Path - What is the shortest path between vertices a and b?
- Cycle - Does the graph contain cycle? diff. algo.
- Connectivity
  - Is given graph connected?
  - Is there a vertex whose removal disconnects the graph?



- Planarity - Can you draw the given graph on a piece of paper without any crossing edges
- Isomorphism - Are two graphs isomorphic?



Graph Problems: Easy to state. Unobvious if the solution is easy, hard or intractable

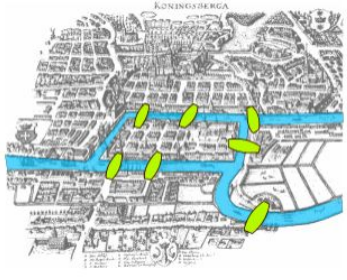
Graph Theory Origin

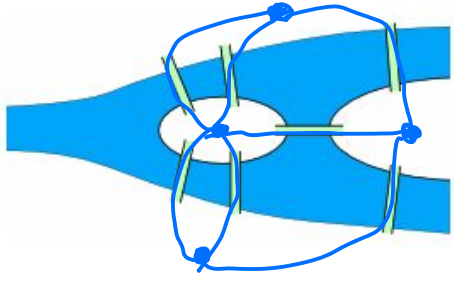
mathmatician, physicisist, astronomer, etc.

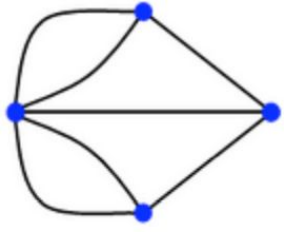
Originated in approx. 1736 when Leonhard Euler asked the Seven Bridges of Kőnigsberg question about the river Pregel in the city of Kőnigsberg where he lived in Prussia. The town had seven bridges crossing the river. He asked:

"Is it possible to walk a route that crosses all seven bridges exactly once and ~~ends up at the same starting point~~?"

Land points ⇒ nodes, bridges ⇒ edges

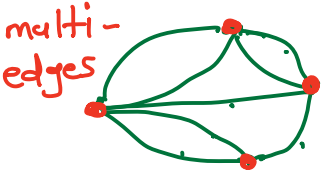




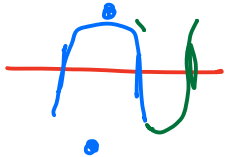


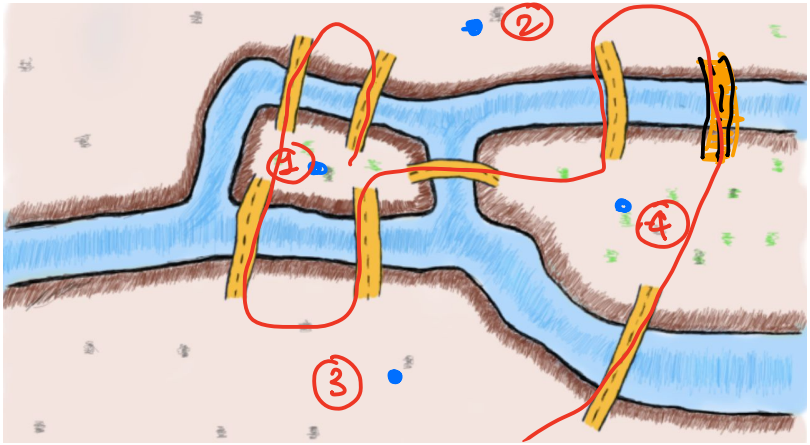
- He solved the problem by representing each section of *land* by a *node* and drawing lines or *edges* between them for each *bridge*.
- He then proved that in general, a walk that *traverses each edge exactly once* and returns to the *starting point* can only exist if each node has an even number of edges leaving it.

If the number of vertices whose degree is odd becomes more than 2, it becomes a problem



odd degree





degree

① = 5

② = ~~2~~ 4

③ = 3

④ = ~~2~~ 3

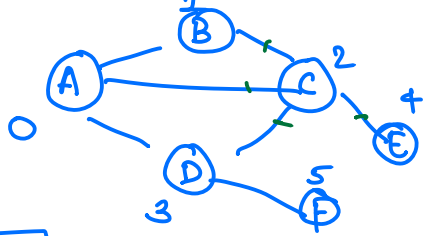
odd degree

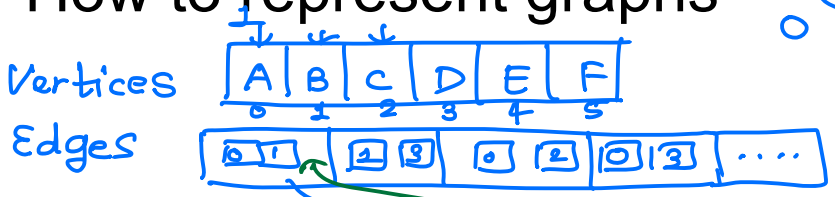
even degree ⇒ Enter and exit the land

odd degree ⇒ Enter the same land twice

degree ⇒ no. of edges that are incident on the vertex

How to represent graphs





Array ?
object in array ⇒ CDT

1 unit of space

Complexity of data structure

Space ⇒ $O(|V| + |E|)$ not bad complexity

Time ⇒ 2 common operations

① ⇒ two nodes adjacent ??

$O(|E|) \approx O(|V|^2)$

quadratic complexity

② ⇒ neighbors of c ??

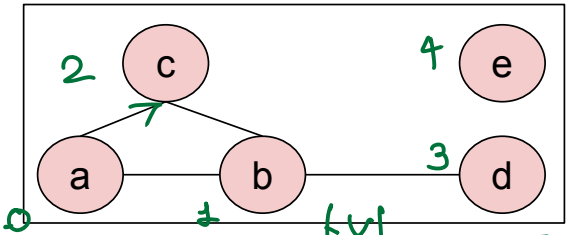
$O(|E|) \approx |V|^2$

Representing Graphs

- Nodes
  - Tracking data items
  - Choices for representation:
    - Arrays, linked lists, trees
- Edges
  - Tracking connection or relationship between nodes
- Two most common ways:
  - Adjacency List:
    - Array of nodes. The i^{th} entry in the array contains a pointer to a linked list that stores the indexes of nodes to which node i is connected.
  - Adjacency Matrix:
    - 2D array of size N × N, where N is the number of nodes in the graph.

Representing Graphs - Adjacency Matrix

- Two vertices are adjacent iff there is an edge between them.



not connected

why not do it for all the graphs?
① setting it up
② doesn't work for directed graph

	a	b	c	d	e
a	0	1	1	0	0
b	1	0	1	1	0
c	1	0	1	0	0
d	0	1	0	0	0
e	0	0	0	0	0

no self loop

|V|

no self loops + Symmetric (only for undirected)

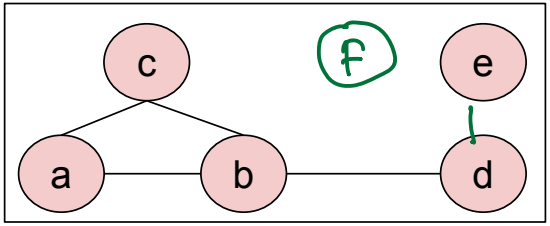
- Advantages:
  - No overhead of edge queries
- Disadvantage:
  - Significant memory wastage

Space Complexity : $O(|V|^2)$
mostly we have sparse graph
Absence wasted memory

time complexity :
① Adjacent (c, b)? $O(1)$
② Neighbors $O|V|$

Complexity - Adjacency Matrix

fixed ⇒ size of adjacency matrix



|V|

	a	b	c	d	e	f
a	0	1	1	0	0	
b	1	0	1	1	0	
c	1	1	0	0	0	
d	0	1	0	0	0	
e	0	0	0	0	1	0
f						

|V|

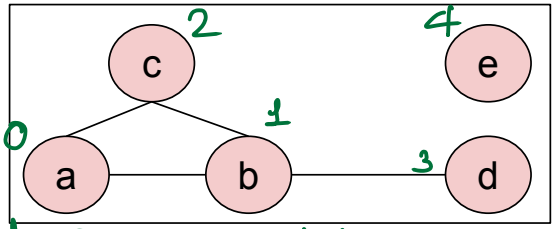
same

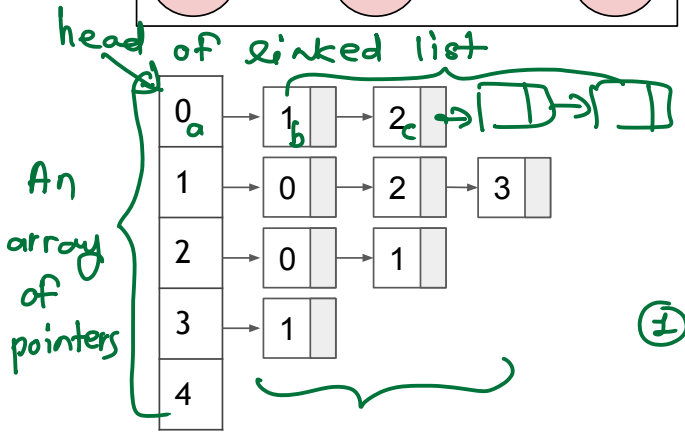
- Are two vertices connected?
  - O(1)
- Are two vertices adjacent?
  - O(1)
- Inserting a node
  - $O(|V|^2)$
- Removing a node:
  - $O(|V|^2)$
- Inserting an edge:
  - O(1)
- Removing an edge
  - O(1)

Representing Graphs - Adjacency Lists

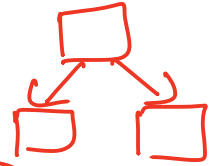
Sparse graphs ⇒ lack of information

- Two vertices are adjacent iff there is an edge between them.





Searching in linked list ⇒ O(|V|)
can we do better? BSTs



- Advantages:
  - Space Efficient ✓
- Disadvantage:
  - Querying an edge requires list traversal

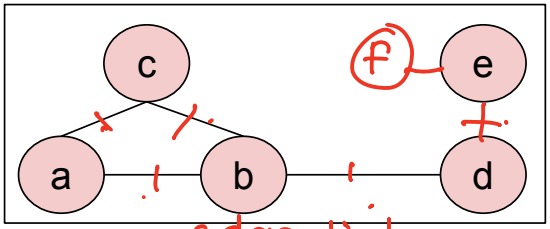
Inserting new node might be a problem

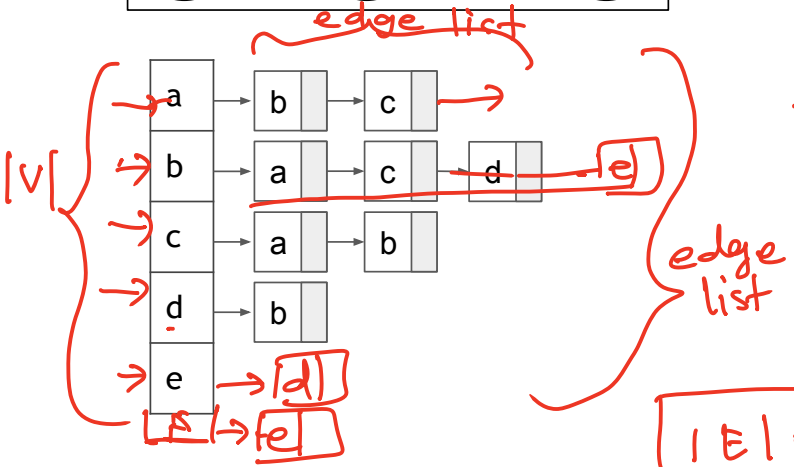
time complexity

① Adjacent ⇒ array of ptrs + list
(2, 1) $O(|V| + |V|) = O(|V|)$

② Neighbors ⇒ (same) $O(|V|)$

Complexity Adjacency Lists





- Are two vertices connected?
  - O(|V|)
- Are two vertices adjacent?
  - ~~O(1)~~ O(|V|)
- Inserting a node
  - O(1)
- Removing a node:
  - O(|E|)
- Inserting an edge:
  - O(1)
- Removing an edge
  - O(|V|)

Assuming that we have space in array of pointers or we have list of pointers

Assuming inserting at the head

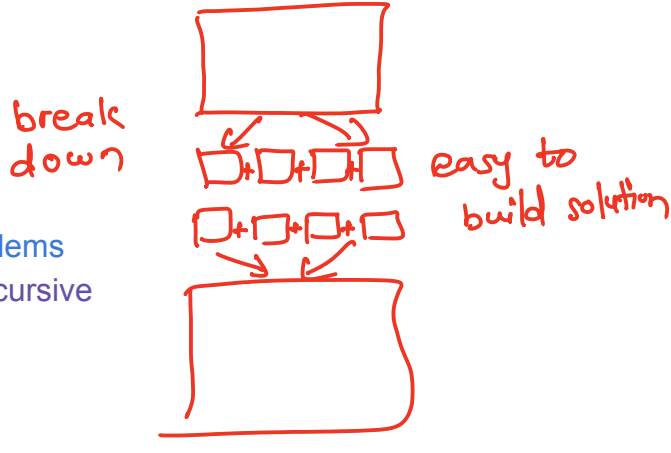
$|E| \leq |V|^2$

Recursion

Recursion - A tool for problem solving

- We have already seen **recursion**!

- Recursion:
  - A way of thinking about problems
  - A way to implement code that solves particular problems
  - A tool for managing data whose structure is itself recursive



A way of thinking about problems

Taking the original input problem ← difficult to solve

    Breaking it up into smaller subproblems

        Breaking those up into even smaller subproblems

            And so on...

                Until the subproblems are so tiny solving the problem is trivial

            We use that solution to solve the slightly larger subproblems

        And then the even larger subproblems

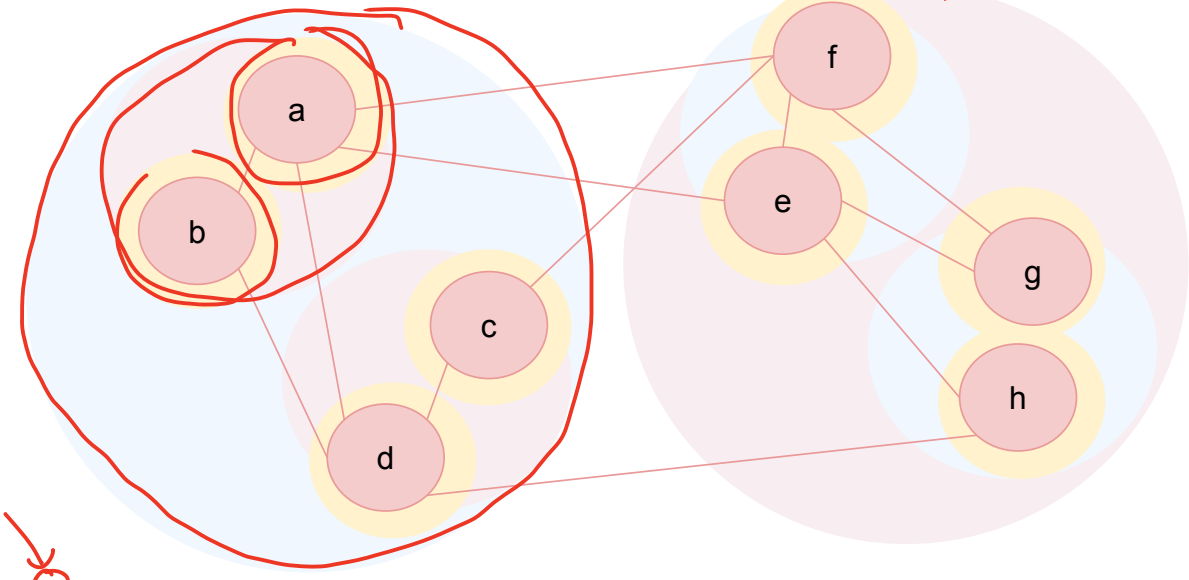
    And then even larger subproblems

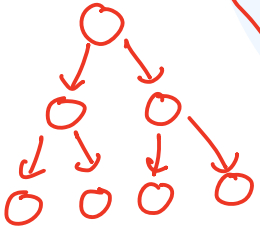
Until we have the solution to the original problem! combine

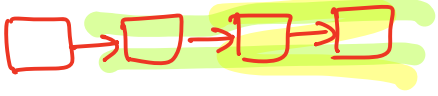
some data structures make it easy to solve problems using recursion

Recursive Data Structures

two more recursive data structures ⇒ BSTs, linked list

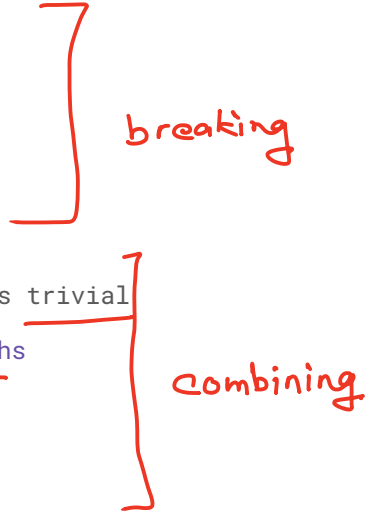






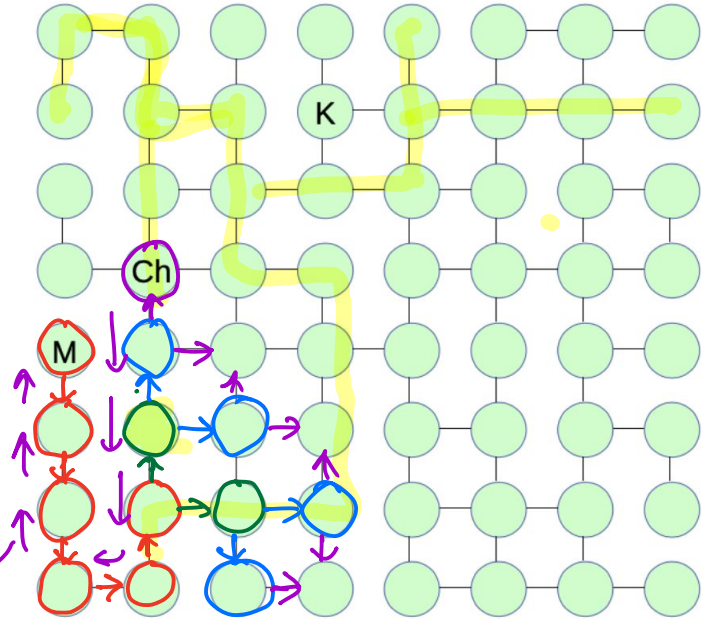
How to take advantage of recursive data structure?

```
Taking the original input graph
    Breaking it up into smaller subgraphs
        Breaking those up into even smaller subgraphs
            And so on...
                Until the subgraphs are so tiny solving the problem is trivial
            We use that solution to solve the slightly larger subgraphs
        And then the even larger subgraphs
    And then even larger subgraphs
Until we have the solution to the original problem!
```



Example from Unit notes

Ask a fiend !! or a neighbour



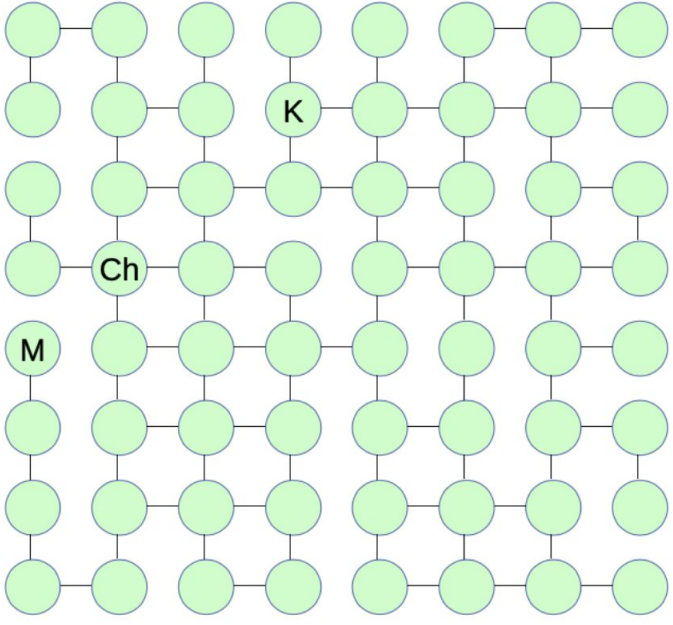
Backtrack

Easy to see which path to take

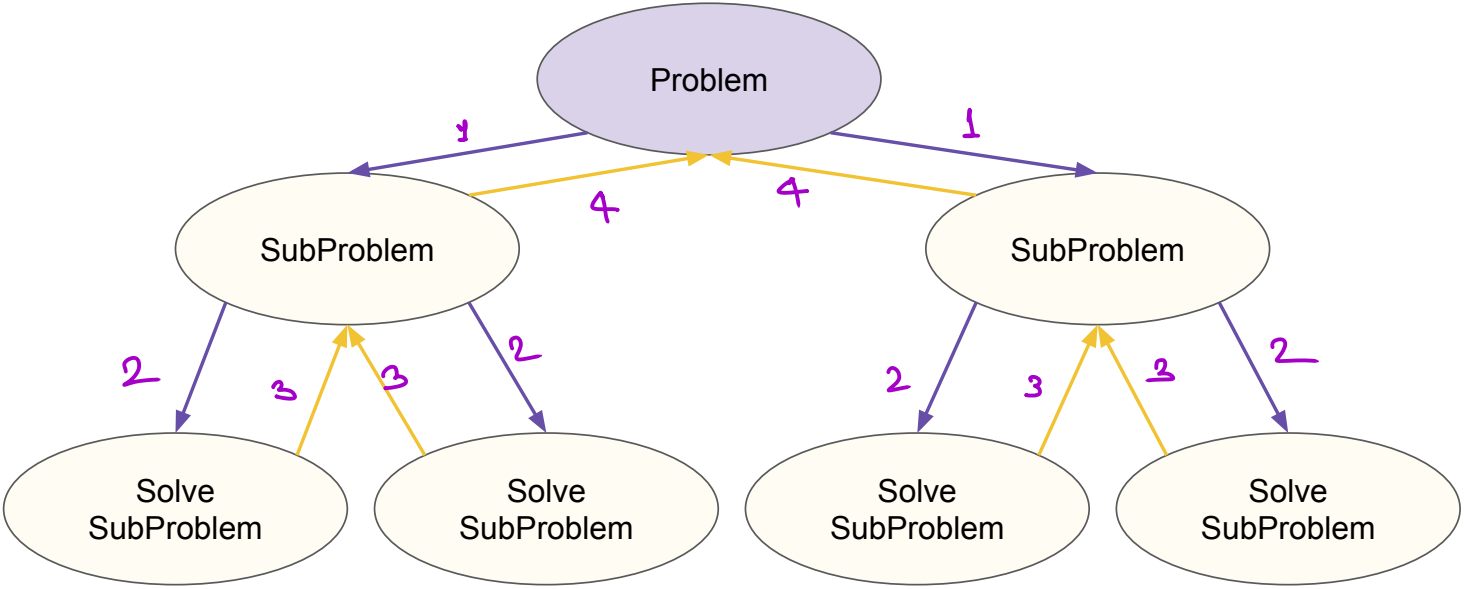
Recursive Approach

```
find path to cheese:
Ask my neighbours: What is the path from you to cheese
    They ask their neighbours: What is the path from you to cheese
        They ask their neighbours ....
            ... until { one of the neighbours-neighbours...neighbours is the cheese!
        Now one of the cheese’s neighbour knows where to go
    Then a cheese’s neighbour’s neighbour knows where to go
... until, the mouse knows where to go!
```

Example from Unit notes



Divide-and-Conquer Strategy



Divide and Conquer Algorithm steps:

- **Divide**: This involves dividing the problem into smaller sub problem.
- **Conquer**: Solving the smaller sub-problems recursively.
- **Combine**: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Applications of Divide and Conquer

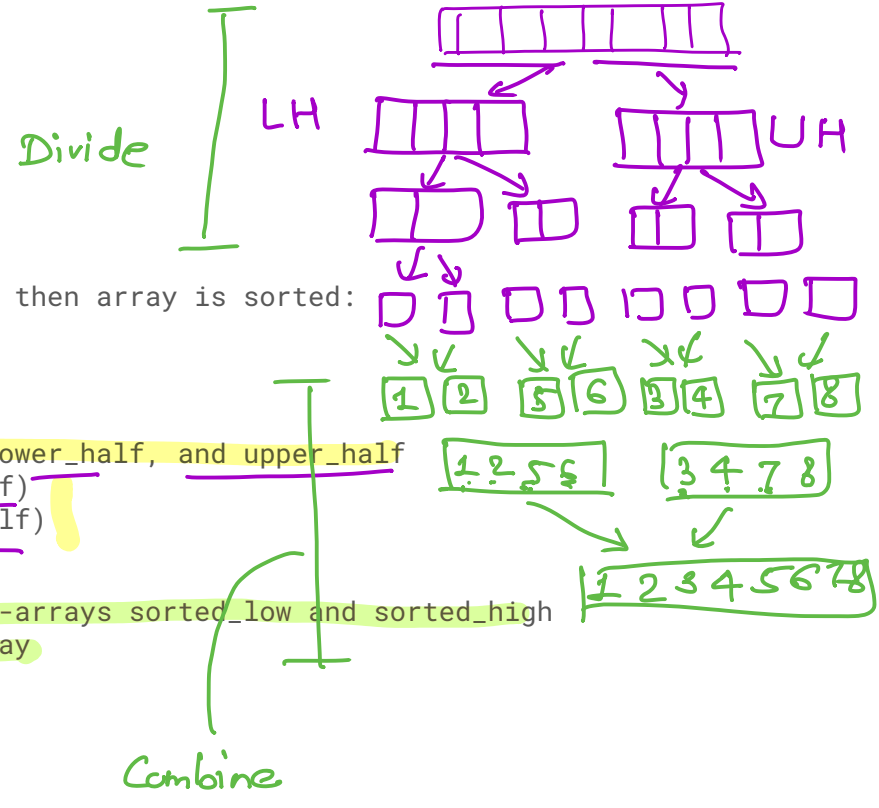
- Binary Search
- Merge Sort
- Quick Sort
- Strassen’s Matrix multiplication

Merge Sort

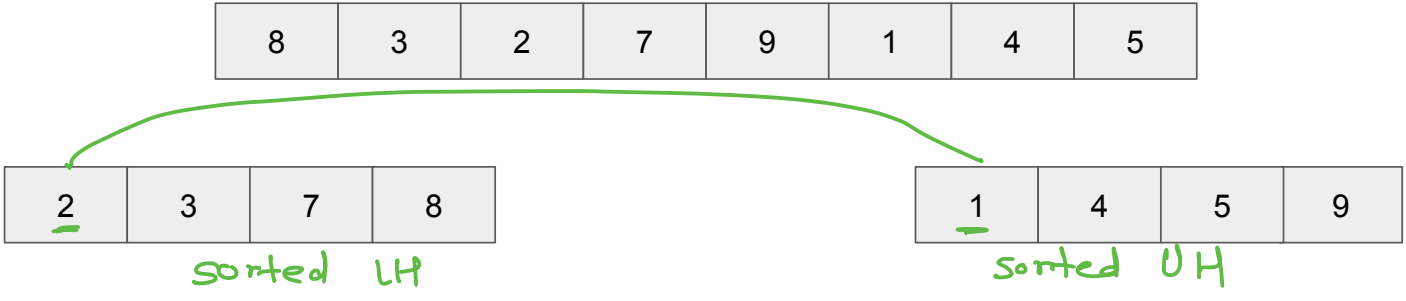
```
mergesort(input_array)

    if the length of input_array is <= 1, then array is sorted:
        return input_array

    else
        split array into 2 sub-arrays: lower_half, and upper_half
        sorted_low = mergesort(lower_half)
        sorted_high = mergesort(upper_half)

        Merge in sorted order the two sub-arrays sorted_low and sorted_high
        to build the complete sorted_array

        return sorted_array
```

dividing

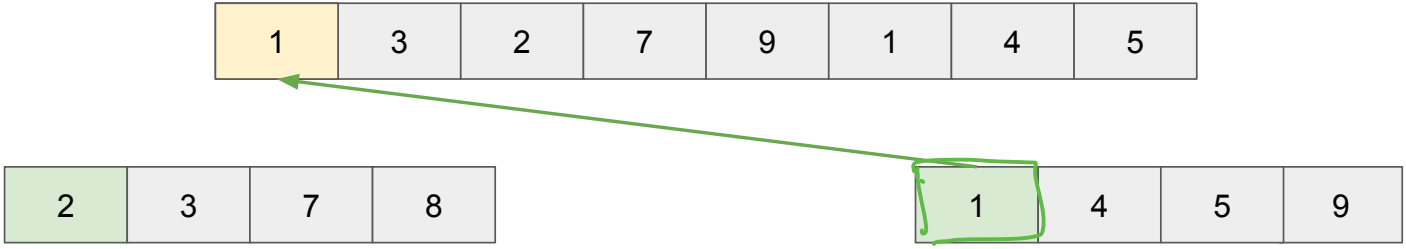
merge



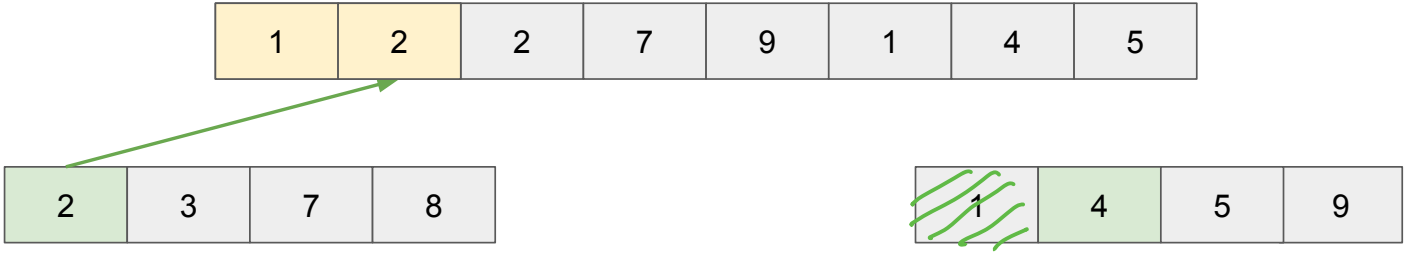
Merge Sort - Array Merging



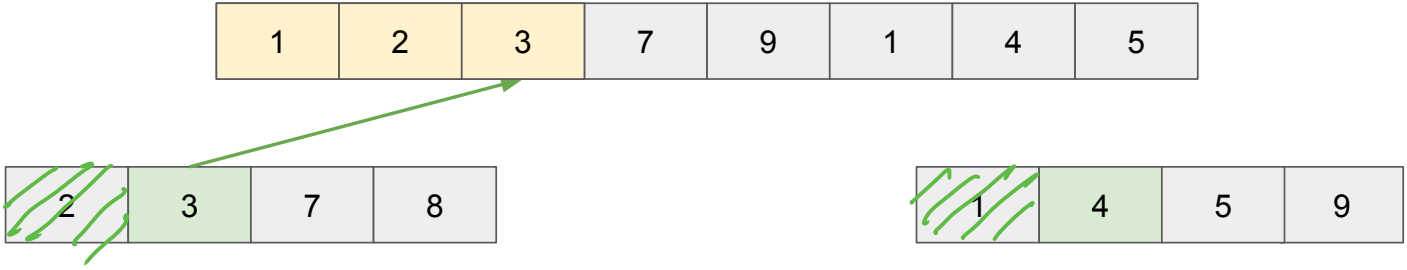
Merge Sort - Array Merging



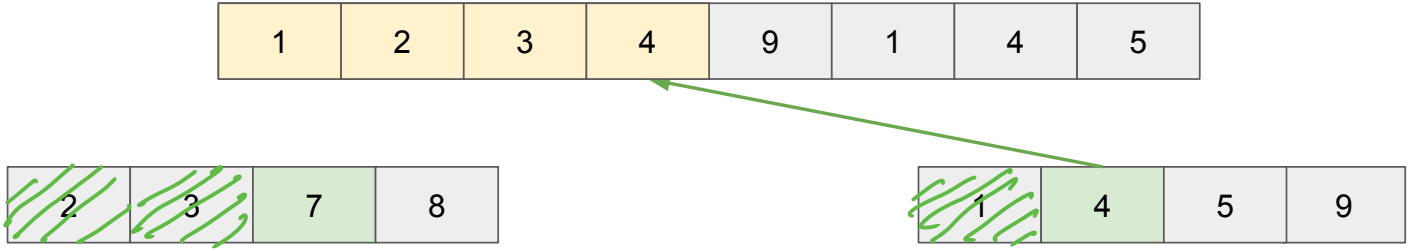
Merge Sort - Array Merging



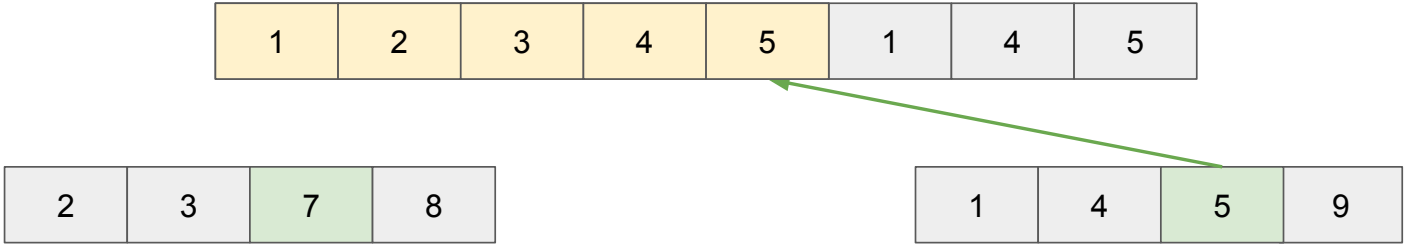
Merge Sort - Array Merging



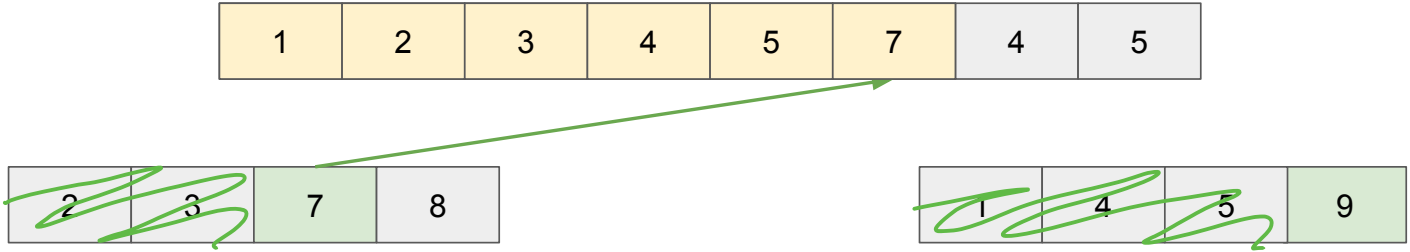
Merge Sort - Array Merging



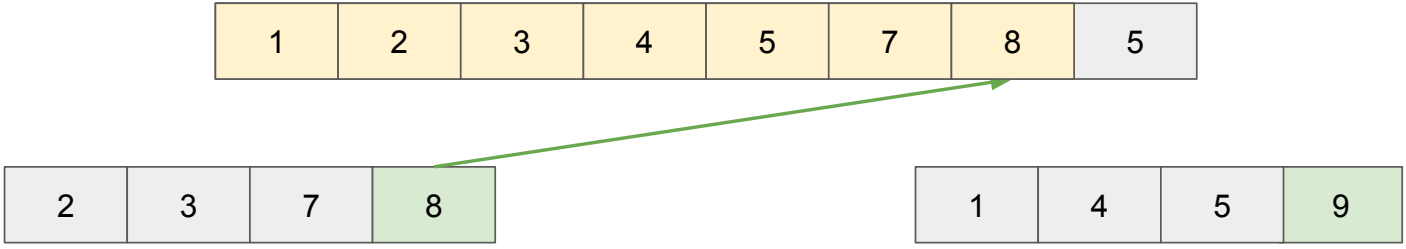
Merge Sort - Array Merging



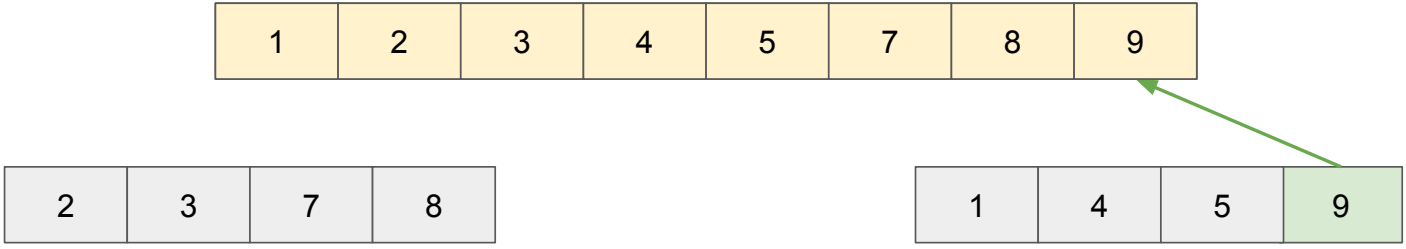
Merge Sort - Array Merging



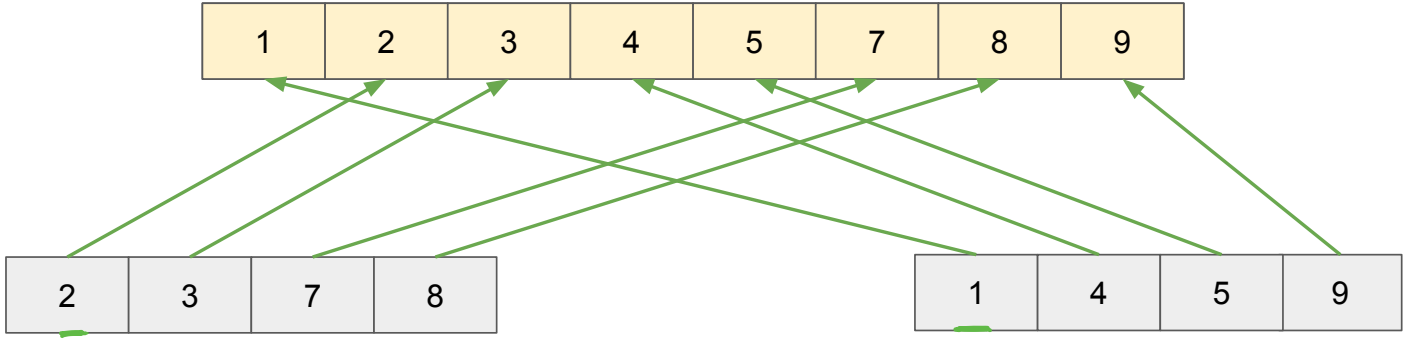
Merge Sort - Array Merging



Merge Sort - Array Merging



Merge Sort - Array Merging

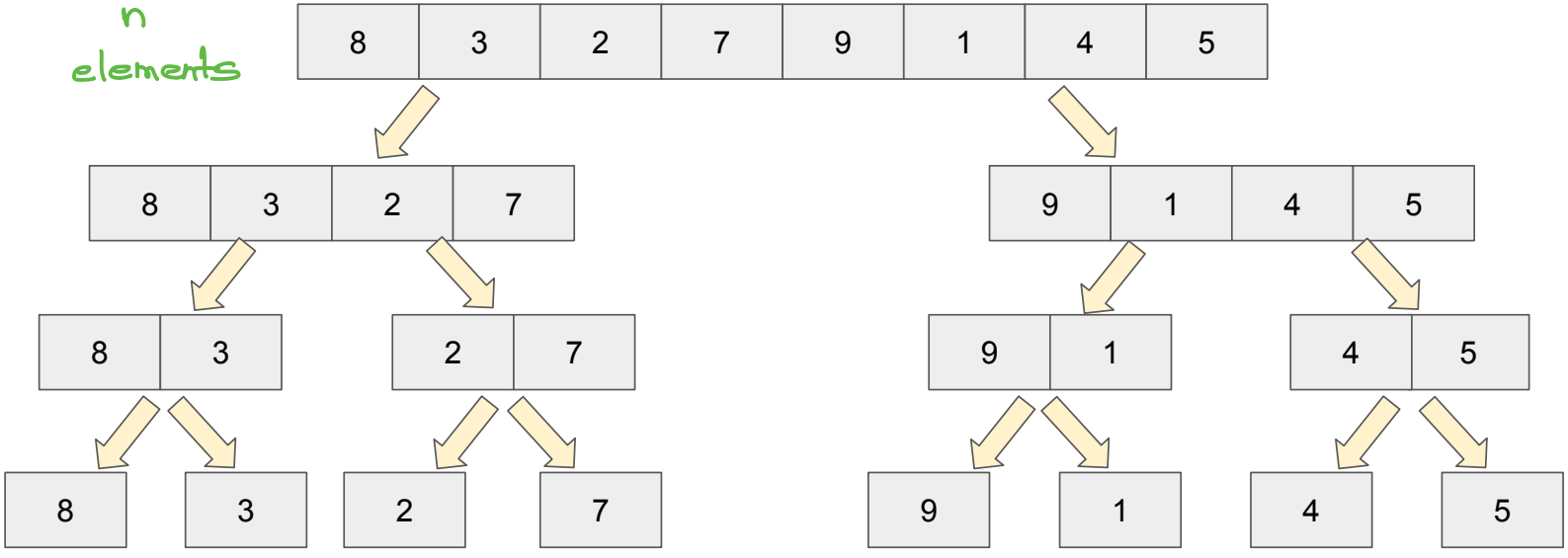


Complexity of array merging for **n** elements??

how many comparisons?

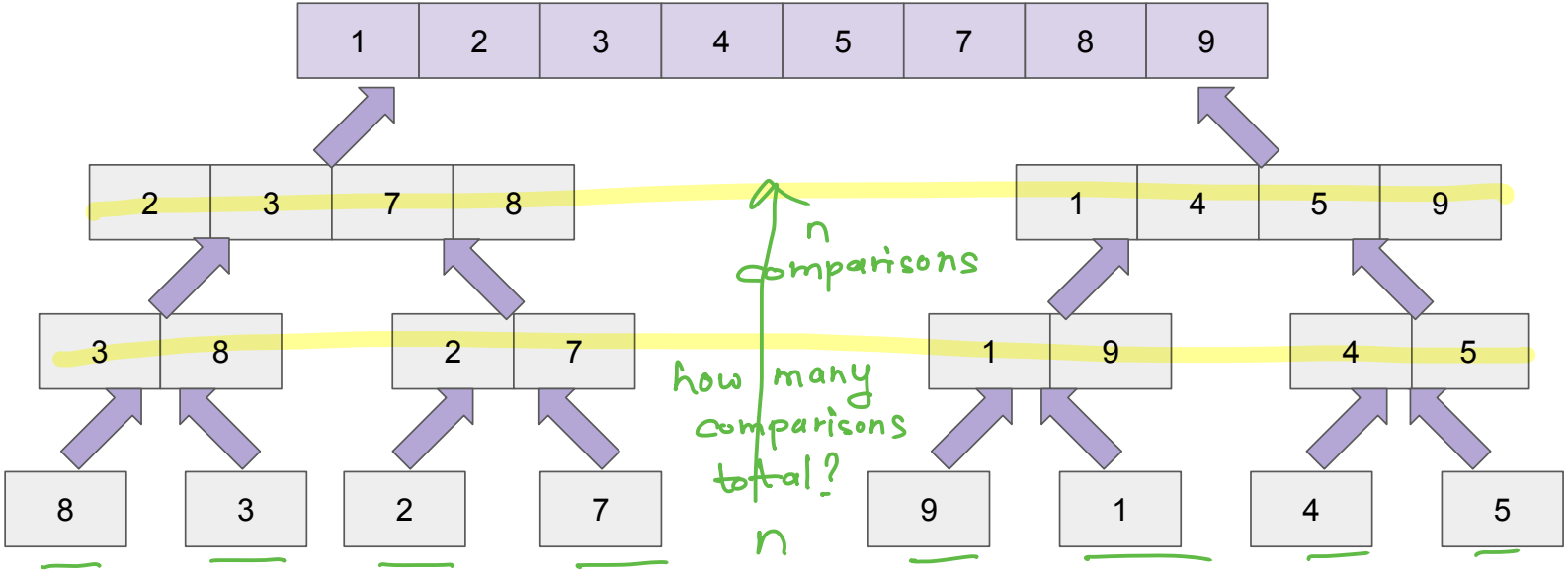
O(1)
O(log n)
O(n) ✓
$O(n^2)$

Merge Sort - Splitting

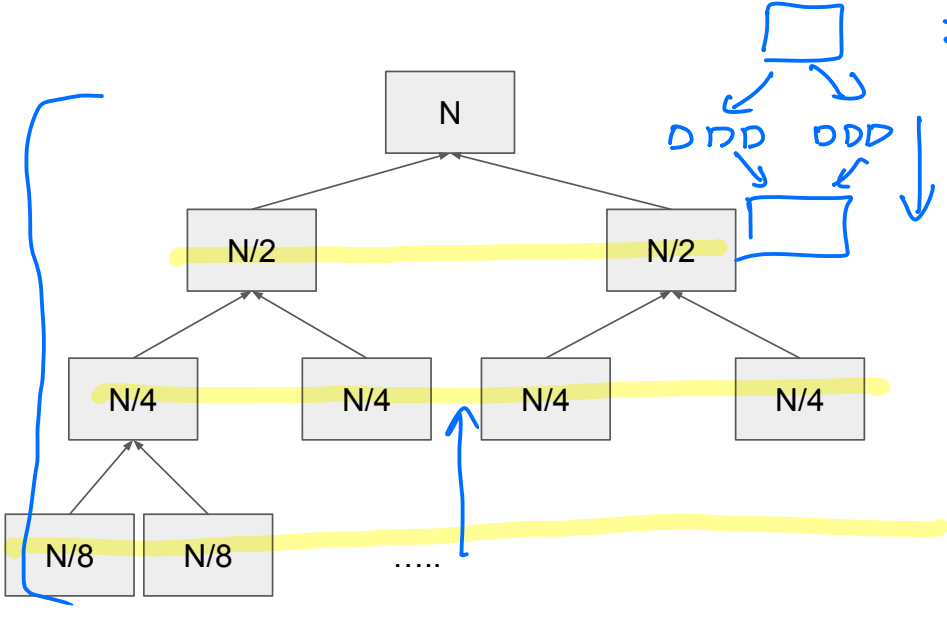


Merge Sort - Merging

Each step ⇒ n comparisons for merge



Merge Sort : Order of Growth



complexity

= time to split
+
time to merge = more work

= no. of levels * amount of work at each level

= $\log_2 N * N$

= $O(N \log_2 N)$

Can there be any disadvantages of recursion??

mergesort ⇒ Extra space $O(n)$

Recursive calls have overhead

all functions

⇓

memory model

1. Recursions are slow & increased time complexity
2. Recursion not suitable for all problems

Set of problems that can be solved using recursion

3. Call stack ⇐ memory requirement