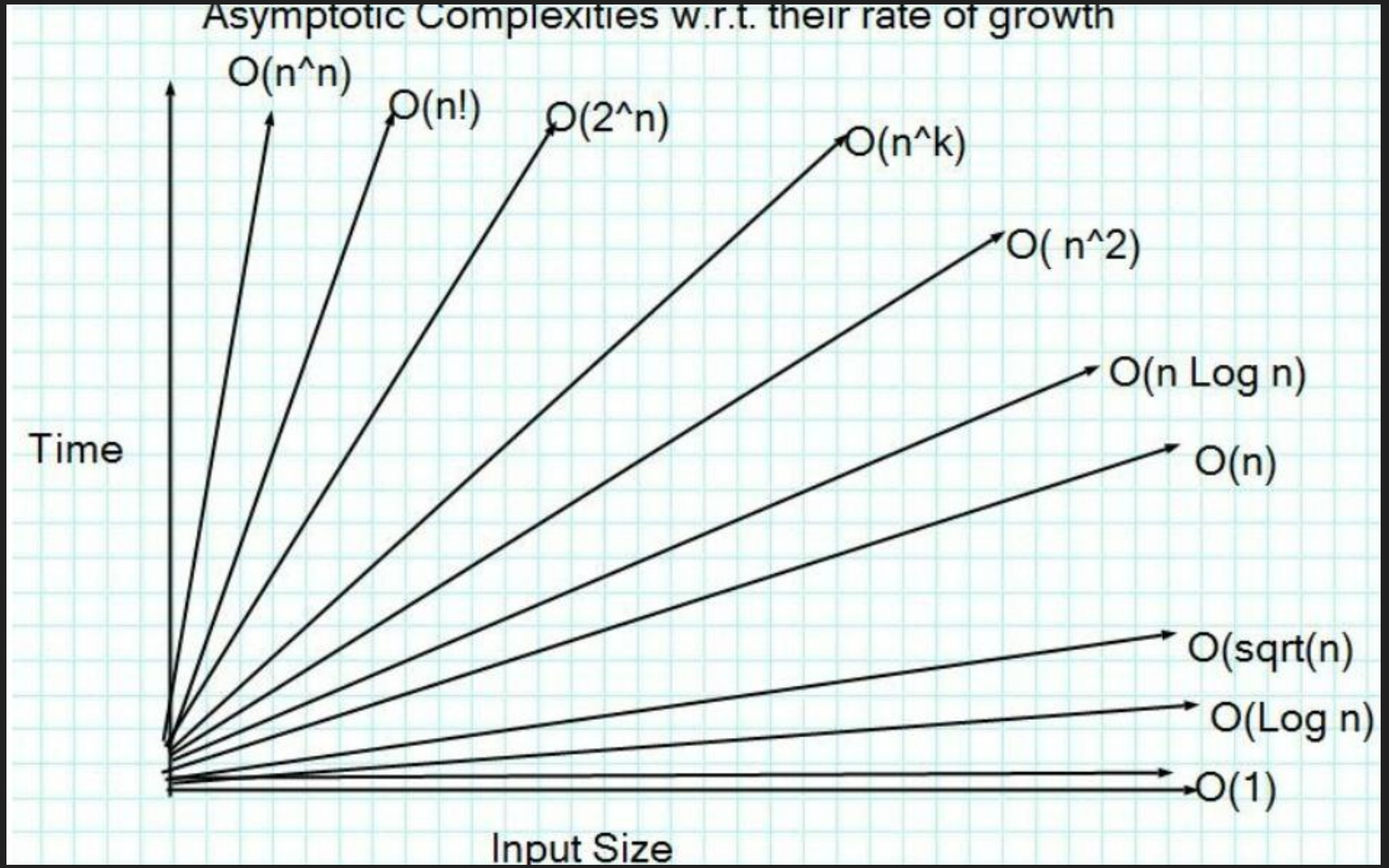
DAA

Vemula Ananya notes


Asymptotic Complexities w.r.t. their rate of growth
O(n^n)
O(n!)
O(2^n)
O(n^k)
O(n^2)
O(n Log n)
O(n)
O(sqrt(n)
O(Log n)
O(1)
Time
Input Size

Searching in an array

- We check if the array exists or not, and then traverse through the array from start to end to find the element K we are looking for. Worst case scenario is that that K element doesn't exist in the array provided. O(n)
- If array were to be sorted, find the midpoint and continue (binary search)
- T(0) = 1 {Immediate answer if the interval of size is 0}
- T(n) = 1 + T(n/2) {Half of the whole array is traversed every time until the value is found}
- T(n) = 1 + T(n/2) = 1 + 1 + T(n/4) = 1 + 1 + …. + T(n/(2^k))
- n/2^k = 1 ; n = 2^k ; k = log n (base 2) ; O(log n)

Binary search code

Smaller sorts

- Selection sort link:
- Selection sort trick: The one with the swapping constantly until the sorted is achieved.
- 0 to 1, 0 to 2, 0 to 3, etc… and then 1 to 2, 1 to 3 etc..
- T(n) = n + (n-1) + (n-2) + (n-3) … +1 = n(n+1)/2 = O(n^2)
- Insertion sort link:
- Insertion sort trick: making a new list and inserting by checking which 2 values the element can go between. Otherwise, keep comparing and swapping with pos and pos – 1.
- T(n) = 1 + 2 + 3 + … + n-1 = n(n-1)/2 = O(n^2)
- Among O(n^2) sorts, the best is insertion sort, then selection sort and then bubble sort
- O(n^2) is infeasible for n over 10,000

Merge Sort

- Merge sort link:
- Merge sort trick: take two sorted lists, make them into stacks, top to bottom is low to high, and then check the topmost element of the stack, compare them and print and pop them. Compare the top of one to the other and repeat until both stacks are empty.
- Take an unsorted array, separate into two, A[0] to A[n/2 -1] and A[n/2] to A[n-1]
- Split the arrays into half again, and again, until there are just one element left.
- Now, merge them all together using the stack method from 1 to 2 to 4
- The merging takes O(n)
- Time taken is 2^jt(n/2^j) + jn
- Overall-> O(n log n)
- Drawbacks: Takes extra memory in order to merge.

Quick Sort

- Quick sort link:
- Quick sort trick: Take two pointers, yellow and green. Make the first element the pivot. Yellow pointer works till the value is less than the pivot, the green one if its greater. Then, if you encounter a smaller in the green or greater in the yellow, swap the locations and change the location of the pointers. Then, put the pivot in the middle by swapping the values. Yellow to the left, pivot in the middle and green on the right. Repeat this process for both yellow and green.
- Option 2) instead of both pointers starting at the index 0, yellow starts at 0, green starts at n-1 and when they interlap, stop the process, and add pivot in them middle.
- As each partition is of size n/2, same as merge sort, t(n) = 2^jt(n/2^j) + jn and average case O(n logn)
- Already sorted array is the worst case being O(n^2)

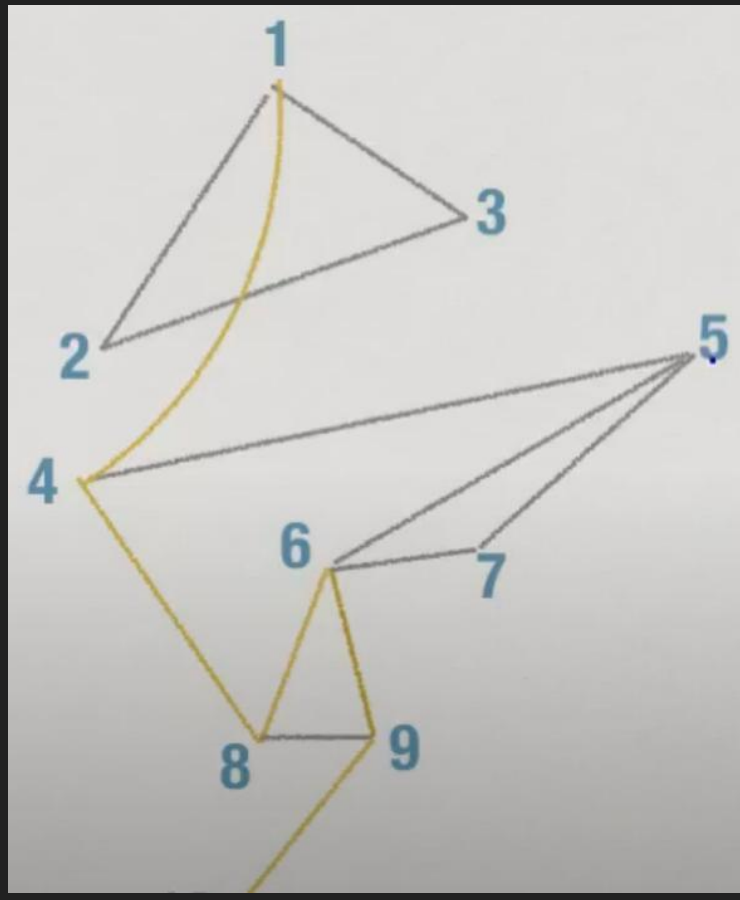
Sorting conclusion

- Sorting needs to be done in a stable manner ie, first priority might be marks of students but then, we need to look at the order of alphabets later in the next step.
- Quick sort is not stable. Merge sort is stable if we merge carefully
- Technically quick sort is best (as long as worst case is handled and pivot is randomized)
- Merge sort is best for external sorting. Merge sort doesn't fit in memory, so disk is used.
- Other O(n log n) algos exist (ex Heap sort)
- Unless data is very small, don't use O(n^2) algos.
- when n <16, use insertion sort.

WEEK 3 - GRAPHS

- Graph colouring is where no two adjacent vertices connected by the edges are of the same colour. In most cases, 4 different colours are enough.
- Graphs consists of a set of vertices and edges.
- In an undirected graph, (v,v') and (v',v) are the same edge. In a directed graph, its not the same as (v,v') means v to v' and (v',v) could or could not exist.
- An adjacency matrix can be used to see which set of vertices are connected.
- Size of matrix is n^2 but if we avoid self loops, maximum size of E is n(n-1)/2
- Create an adjacency list to maintain the list of all the neighbours per vertex.

Adjacency matrix



	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0	0
4	1	0	0	0	1	0	0	1	0	0
5	0	0	0	1	0	1	1	0	0	0
6	0	0	0	0	1	0	1	1	1	0
7	0	0	0	0	1	1	0	0	0	0
8	0	0	0	1	0	1	0	0	1	0
9	0	0	0	0	0	1	0	1	0	1
10	0	0	0	0	0	0	0	0	1	0

1	2,3,4
2	1,3
3	1,2
4	1,5,8
5	4,6,7
6	5,7,8,9
7	5,6
8	4,6,9
9	6,8,10
10	9

Breadth First Search

- BFS or Breadth first search is done using queue() you keep a visited set as well to mark the nodes which have been visited.
- We refer to the queue, once the neighbours of the element in the queue have been visited, we pop the element and move on to the next, if the neighbour of the next element are already in the visited, then we move onto the next element in the queue.
- That way, we can find the element we are looking for. If all are visited, the element is missing
- Complexity with an adjacency matrix is O(n^2)
- At the same time, if adjacency list is provided, O(n+m)
- One way to keep track of the previous nodes is by using parent[k] = j
- Otherwise, keep track of level. Ie, level = -1 initially and level[i] =p means its p steps from I
- Therefore, To search: visited; To find shortest/longest path: level; To track paths: parent
- BFS is good for unweighted paths, Djkstras for weighted graphs, to find the shortest paths.

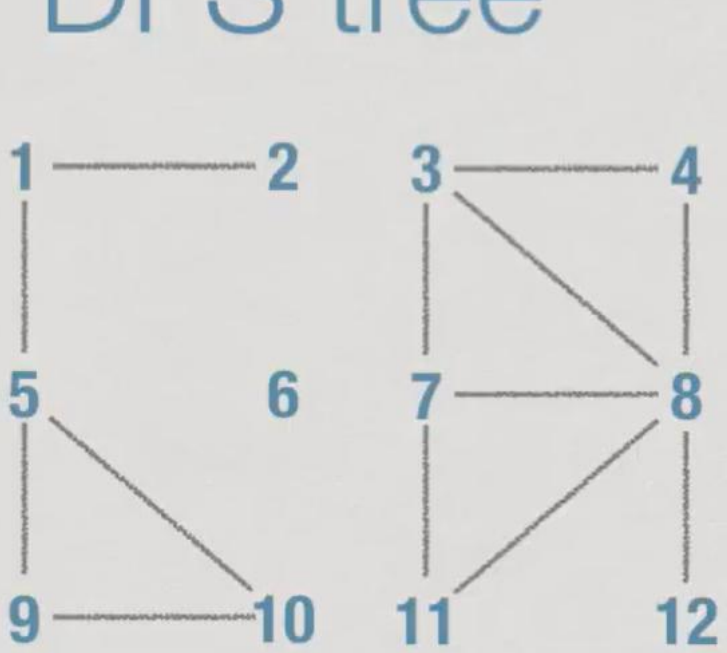
Depth First Search

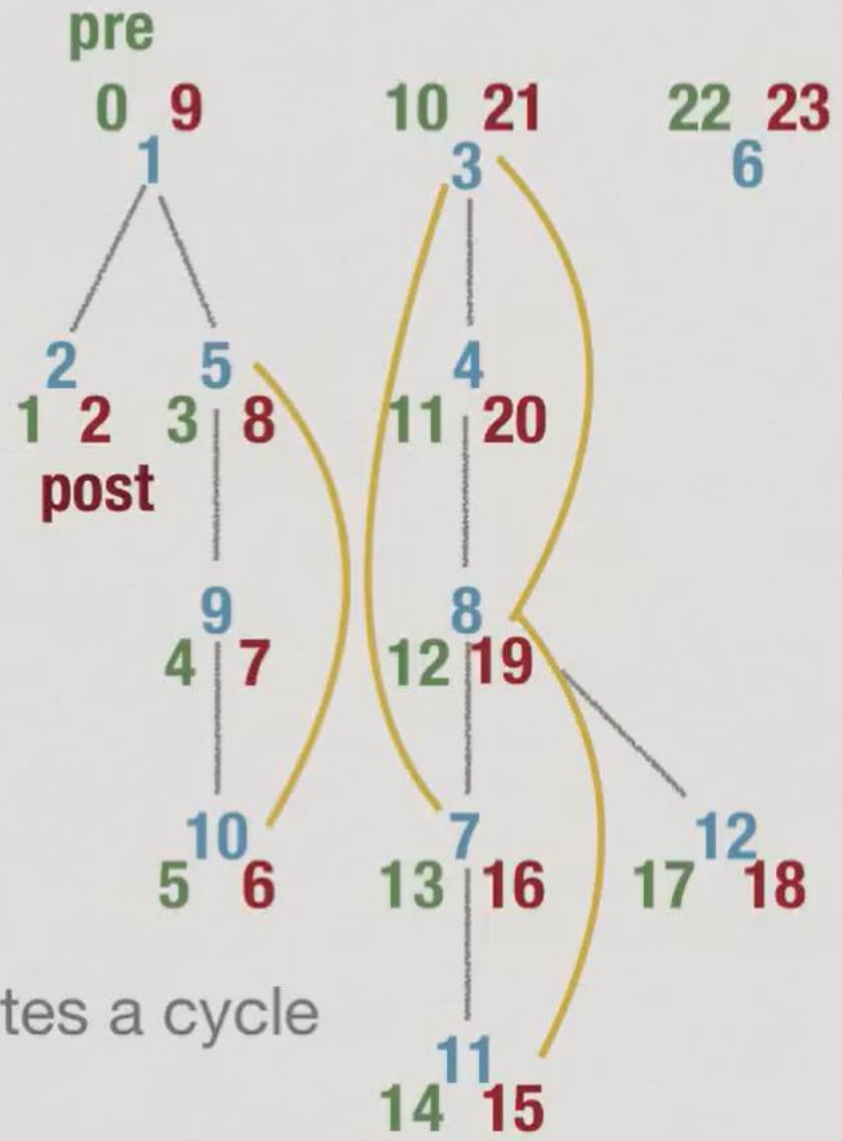
- DFS or Depth first search is done using stack. you keep a visited set as well to mark the nodes which have been visited.
- We start at a node and check for the neighbours, then we pick one of the neighbours and look at its neighbours, all whilst keeping track of the visited nodes. We backtrack in case all the neighbours of the nodes are visited and repeat the process.
- Once all the possible nodes are visited, we pop all the nodes still in the stack and stop the processes.
- We explicitly don't need to maintain a stack as its done by the recursion process already.
- Complexity with an adjacency matrix is O(n^2)
- At the same time, if adjacency list is provided, O(n+m)
- Create a post[i] and pre[i] to find cut vertices and loops in the provided graph

Applications of BFS and DFS

- To check for connectivity of graphs or number of provinces, we can perform DFS or BFS on each node until no other node can be reached. Then, we can go to another node and continue the same process while keeping track of count of number of provinces. (as given in NPTEL, comp or component[j] = comp)
- Next thing to check for is cycles. As BFS forms a tree, when performed BFS on a cyclic graph and number of edges are calculated, the number of edges calculated will be less than the actual number of edges, hence proving the graph is cyclic. The non tree edges are the edges that end up forming cycles.
- Strongly connected paths, if there is a path from i to j and j to i, they are strongly connected. DFS numbering using pre and post can be used to compute SCC's
- Other structural properties alike articulate points (vertices) removing such vertex disconnects the graph and Bridges(edges) removing such edges disconnects the graph.

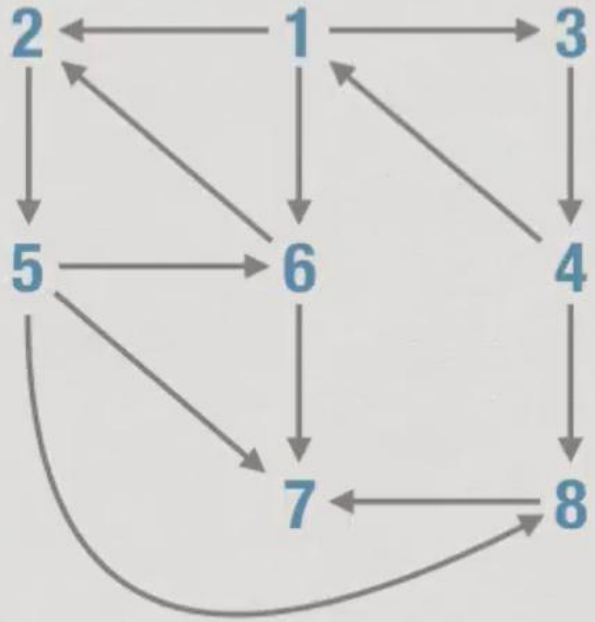
DFS tree

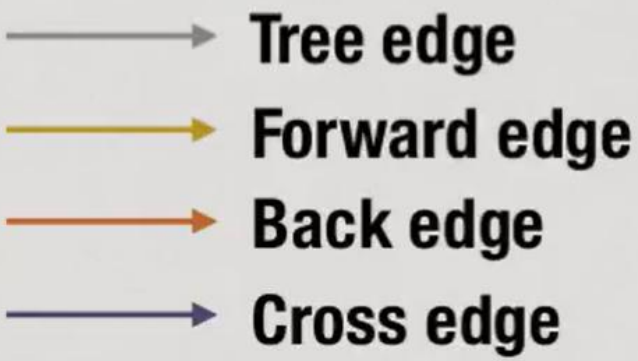


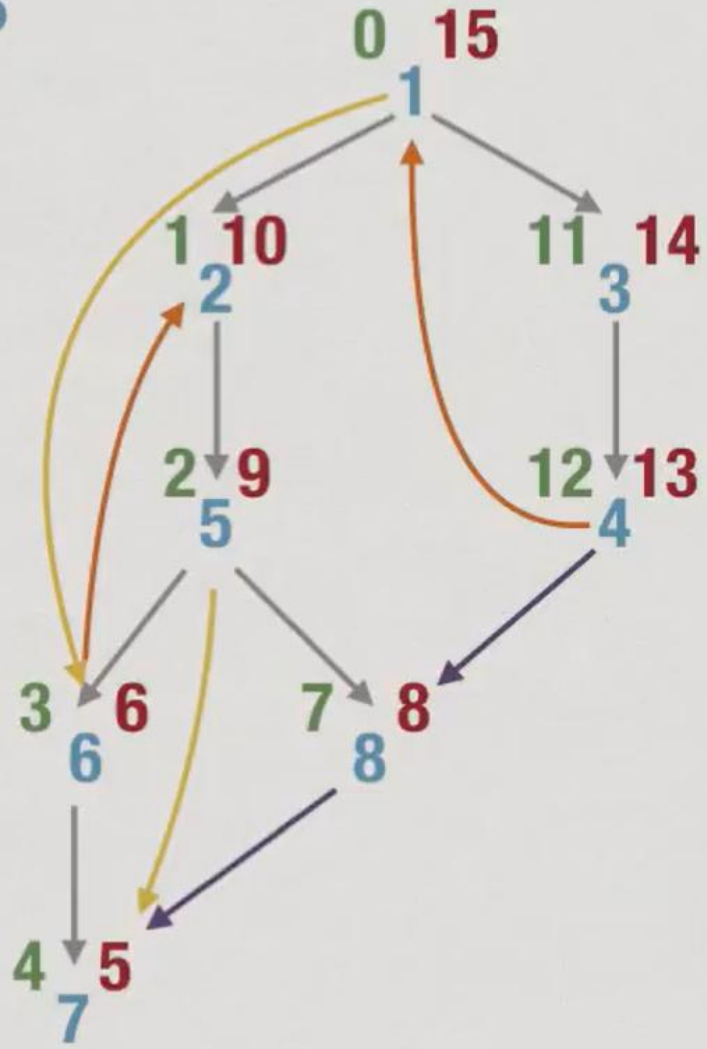


- Any non-tree edge generates a cycle

Directed cycles







- Cross Edges go only from higher numbers to lower numbers (13 to 8 to 5)
- Also, cycles only form where there are back edges.

- A directed graph has a cycle if and only if DFS reveals a back edge
- Can classify edges using pre and post numbers
  - Tree/Forward edge (u,v) :
    Interval [pre(u),post(u)] contains [(pre(v),post(v)]
  - Backward edge (u,v):
    Interval [pre(v),post(v)] contains [(pre(u),post(u)]
  - Cross edge (u,v):
    Intervals [(pre(u),post(u)] and [(pre(v),post(v)] disjoint

- Forward edge example [0,15], [1,10]

Here, [1,10] lies between [0,15]

Ie, [u u]

[v, v]

- Backward edge example [3,6], [1,10]

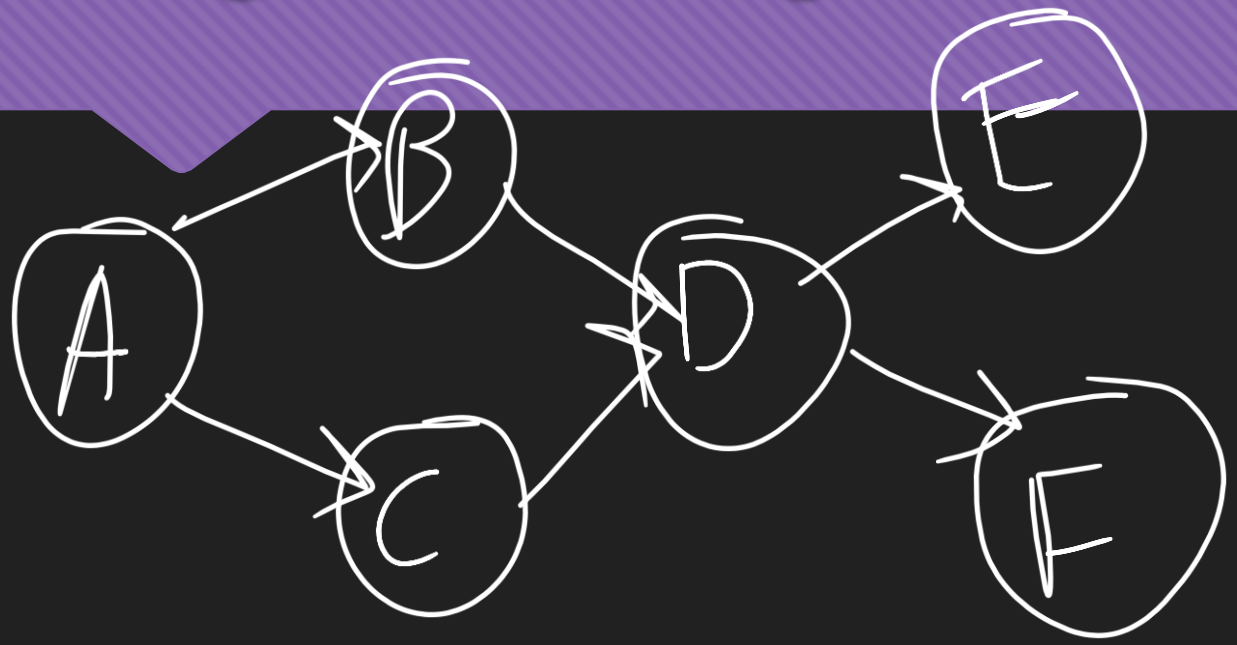
Here, [3,6] lies between [1,10]

Ie, [u u]

[v v]

- Cross edges are disjoint

Topological sorting



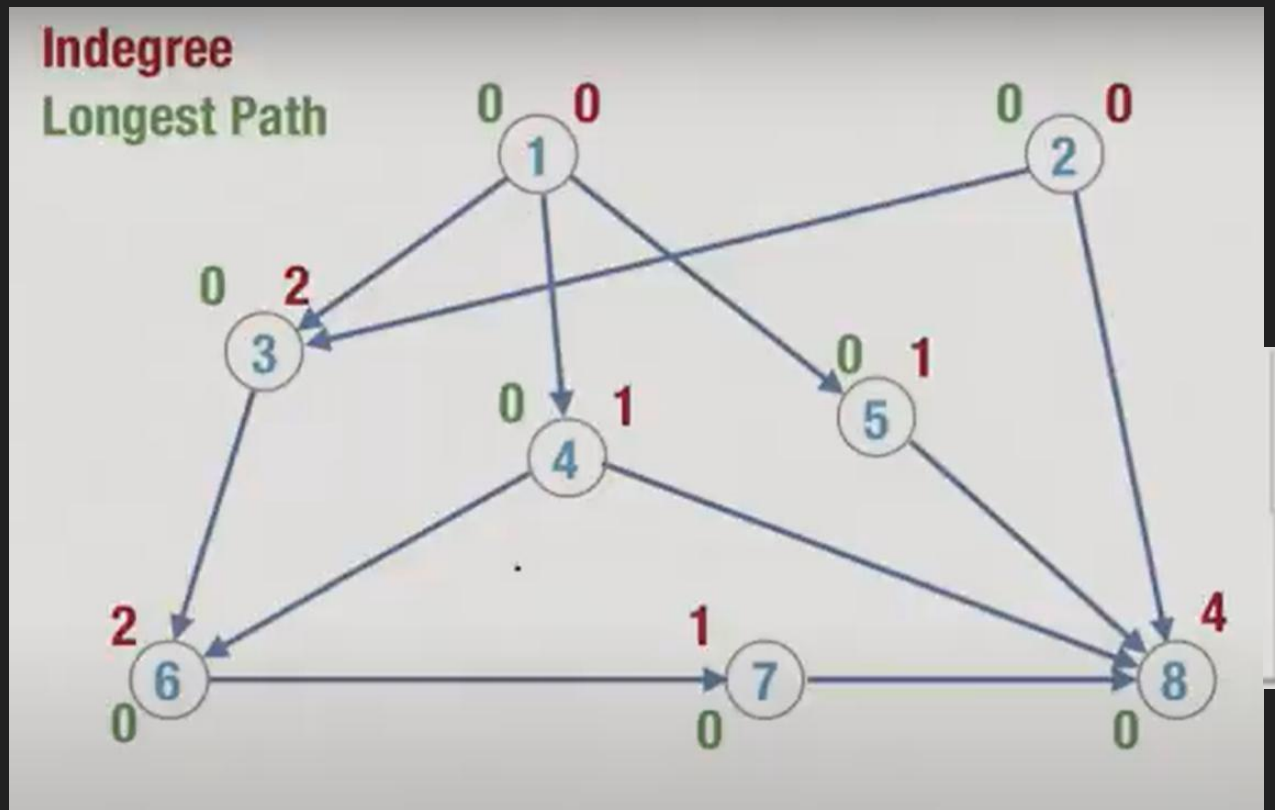
- First step is A, second either B or C then D then either E or F but every node is to be visited.
- Ex) A,B,C,D,E,F or A,C,B,D,E,F or A,B,C,D,F,E or A,C,B,D,F,E
- Directed acyclic graphs like these are called DAGs
- Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.

Topological sorting

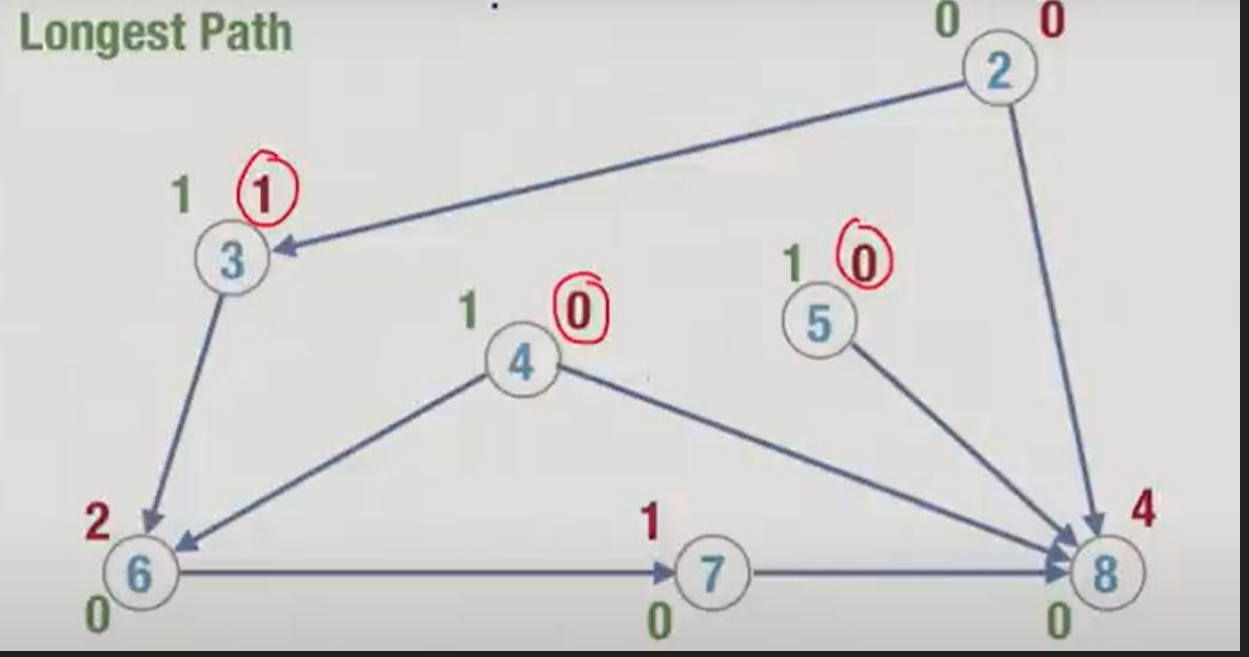
- Indegree(v) the number of edges into v
- Outdegree(v) is the number of edges out of v
- Every DAG has at least 1 vertex with indegree 0
- Pick a verted of indegree 0, remove that node from the graph (ie, subtract 1 from the indegrees of its neighbours)
- A new set of indegrees will form(only for the once connected) pick with indegree 0 and continue the process until the whole graph disappears
- If there is no node with indegree 0, that means cycle exist in the graph
- Using adjacency matrix, complexity = O(n^2)
- Using adjacency list, complexity = O(m+n)

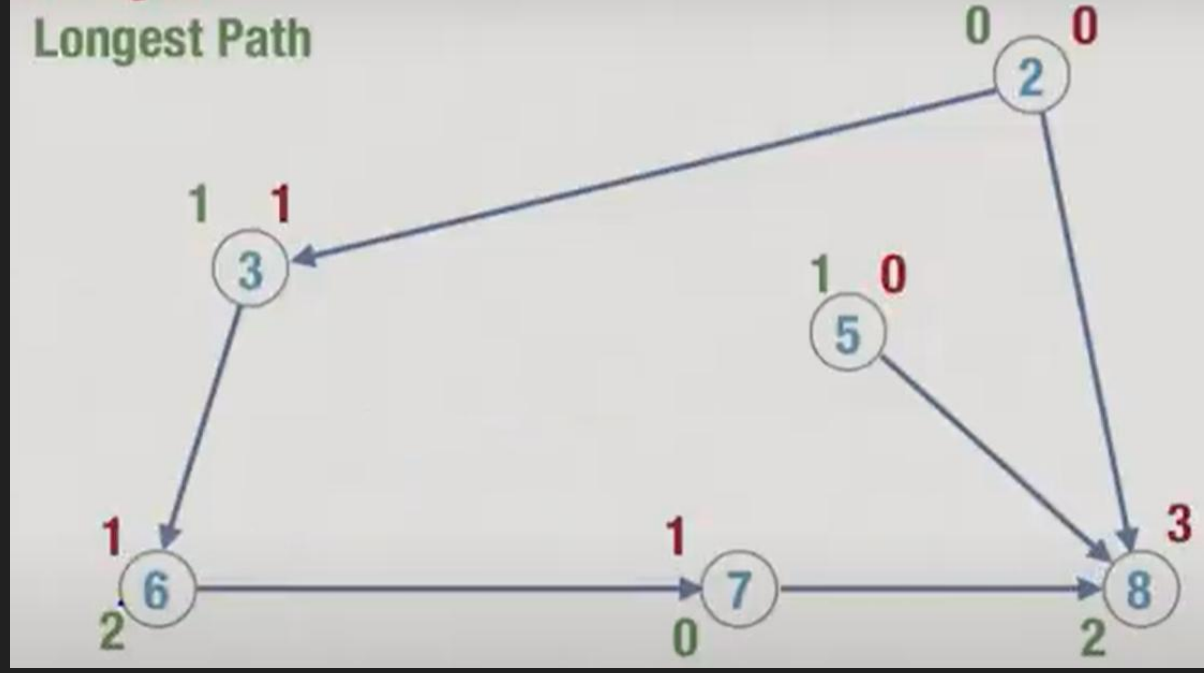
Topological sorting

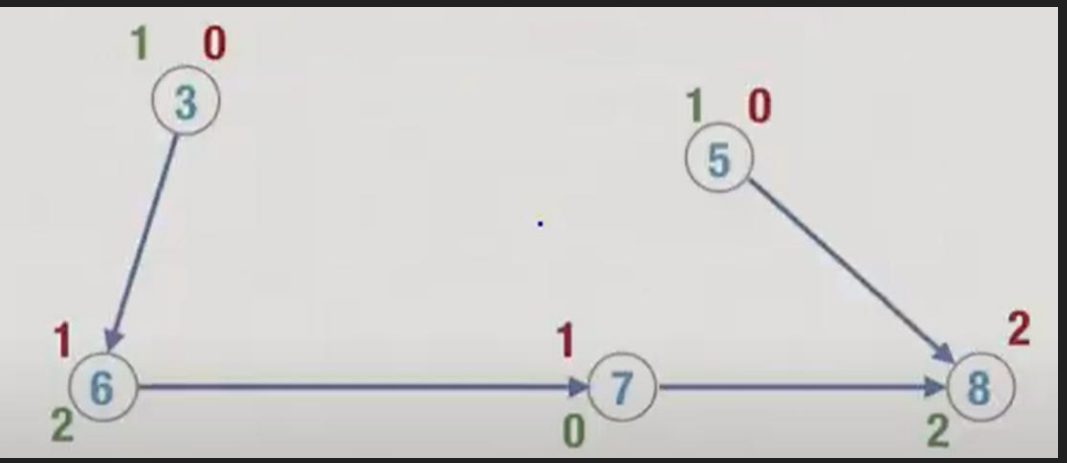
- To find the longest path in the DAG, if indegree(j) = 0, longest_path_to(j) = 0. If indegree(k) > 0, longest_path_to(k) is 1 + max{longest_path_to(j)} among all incoming neighbours j of k

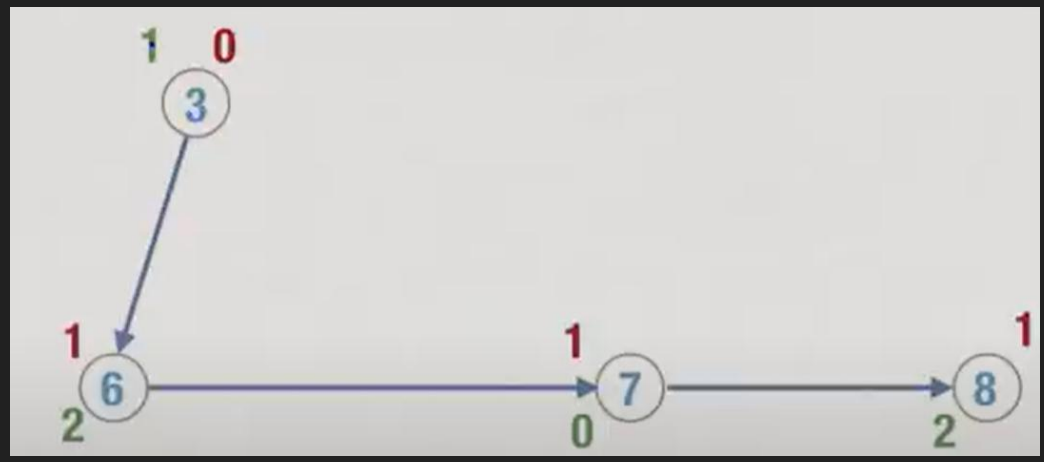


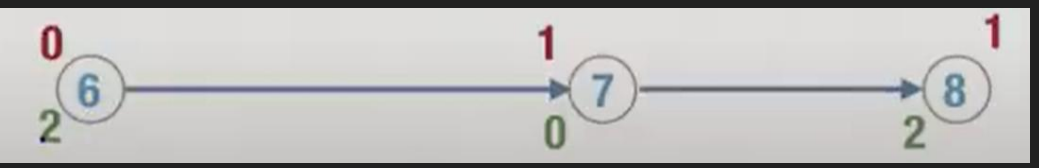
1	4	2	5	3	6	7	8
0	1	0	1	1	2	3	4

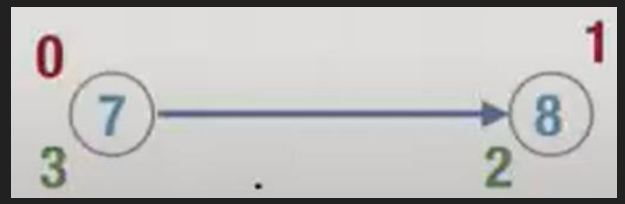

Longest Path
0 0
2
1 1
3
1 0
5
1 0
4
2
6
1
7
4
8
0
0
0

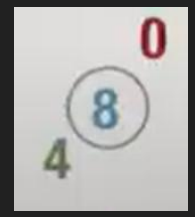

Longest Path
0 0
2
1 1
3
1 0
5
1
6
1
7
3
8
2
0
2


1 0
3
1 0
5
1
6
1
7
2
8
2
0
2


1 0
3
1
6
1
7
1
8
2
0
2

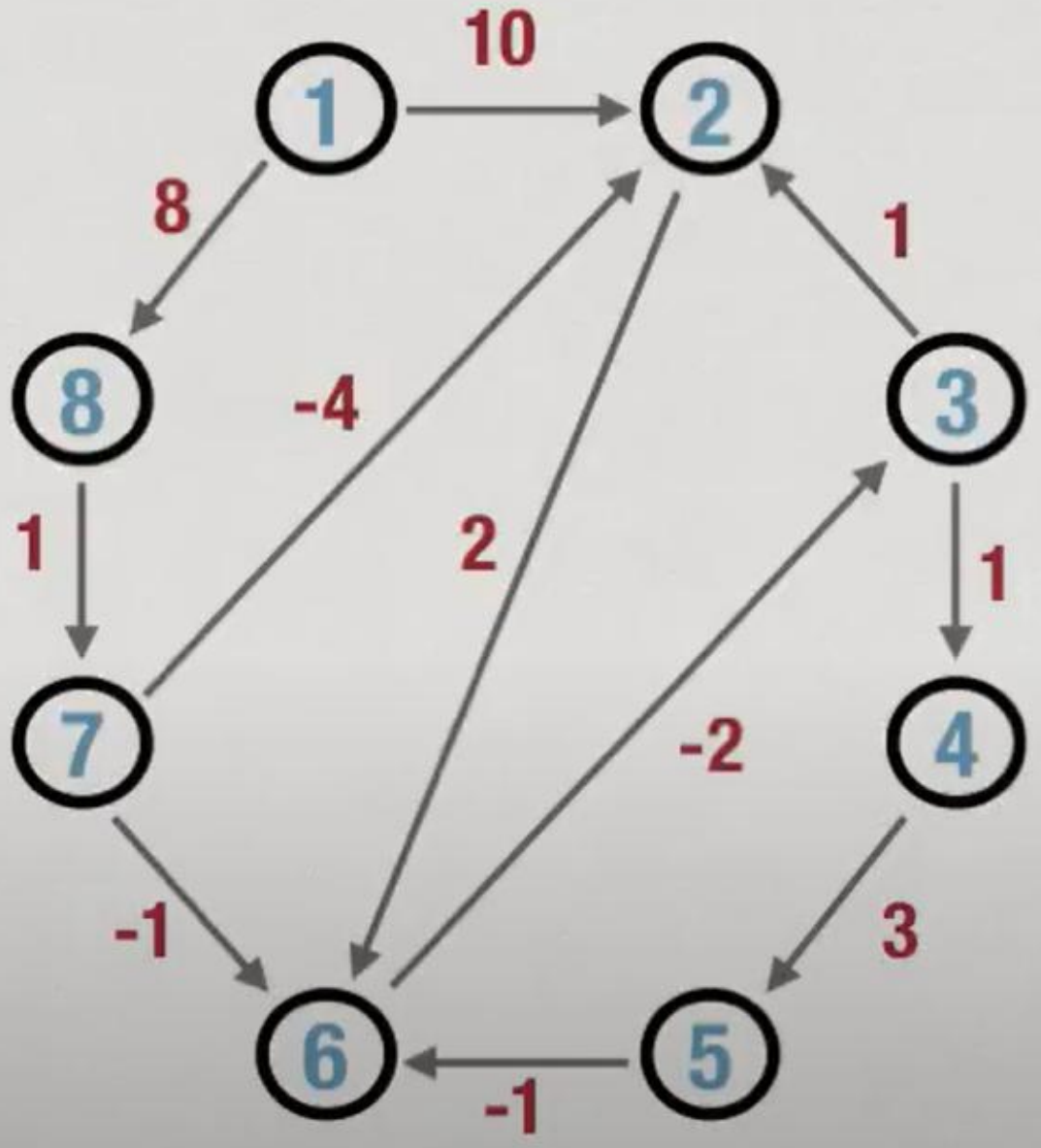

0
6
1
7
1
8
2
0
2


0
7
1
8
3
2


0
8
4

Dijkstras algorithm

- BFS find the path with the fewest number of edges, need not be the shortest path
- Greedy algorithm where the next choice depends on the current best value and doesn't go back to change the choice.
- Current best choice is not always optimal.
- With the help of adjacency matrix, outer loop runs n times to burn a vertex with the minimum weight. The inner loop runs to scan all the neighbours and update the burn time. $O(n^2)$
- Using adjacency lists, the outer loop and inner loop run and the whole computation is O((n+m)log n)
- Bellman-Ford is used to determine shortest paths with negative weights but no negative cycles.
- Distance(k) = min(Distance(k), Distance(k) + weight(j,k))
- https://www.youtube.com/watch?v=9PHkk0UavIM



Iteration

Node	0	1	2	3	4	5	6	7
1	0	0	0	0	0	0	0	0
2	∞	10	10	5	5	5	5	5
3	∞	∞	∞	10	6	5	5	5
4	∞	∞	∞	∞	11	7	6	6
5	∞	∞	∞	∞	∞	14	10	9
6	∞	∞	12	8	7	7	7	7
7	∞	∞	9	9	9	9	9	9
8	∞	8	8	8	8	8	8	8

Dijkstras algorithm

- Floyd- Warshall algo creates time complexity to be O(n^3) and space complexity to be O(n^2).
- $W^k(i,j) = \min(W^{(k-1)}(i,j), W^{(k-1)}(i,k) + W^{(k-1)}(k,j))$
- Unless direct path mentioned in the graph (direct paths), assume infinity (Unlike bellman-ford, the rows and columns both represent nodes and not iterations!)
- W^0 and W^1 are always usually same.
- Repeat this process until the shortest path is found.
- Complexity is $O(n^3)$. Say slices are used (from I,j,k-1 to I,j,k) the complexity reduces to $O(n^2)$
- Floyd-Warshall originally proposed an algo for transitive closure.
- https://www.youtube.com/watch?v=4OQeCuLYj-4

W^0

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	3	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^1

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	∞	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	∞	∞	∞
4	∞	∞	3	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-1	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^2

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	12	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	3	∞	∞
4	∞	∞	3	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	∞	-2	∞	∞	∞	∞	∞
7	∞	-4	∞	∞	∞	-2	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

W^3

	1	2	3	4	5	6	7	8
1	∞	10	∞	∞	∞	12	∞	8
2	∞	∞	∞	∞	∞	2	∞	∞
3	∞	1	∞	1	∞	3	∞	∞
4	∞	∞	3	∞	∞	∞	∞	∞
5	∞	∞	∞	∞	∞	-1	∞	∞
6	∞	-1	-2	-1	∞	1	∞	∞
7	∞	-4	∞	∞	∞	-2	∞	∞
8	∞	∞	∞	∞	∞	∞	1	∞

Trees

- A Tree of n nodes has n-1 edges.
- Adding an edge in the tree creates a cycle.
- In a tree, every pair of nodes is connected by a unique path
- Minimum cost spanning tree works on Prims algo and Kruskal's algo.
- Prims algo: start with the smallest edge and grow it into a tree
- Kruskal's algo: scan edges in ascending order of cost and connect them to form a tree

Prims algo:

- Greedy algorithm. Local heuristic decides next step.
- Choices made are never reconsidered.
- For any vertex v, The smallest edge attached to it should be in the spanning tree.
- Adjacency matrix, O(n^2). Adjacency list O((n+m)log n)

Trees

Kruskal algo:

- Order edges in ascending order by weight
- Keep adding edges to combine components
- With adjacency matrix, O(n^2). Adjacency list, O(m log n)

Union-find using arrays:

Step1) Set up an array component[1…n]

MakeUnionFind(s): set compoenet[i] = i for all I

Find(i) to return component[i]

Union(k,k'): For each 1…n, for component[i] == k, component[i] = k'

Complexity is O(mn) or O(m^2)

Trees & Union Find

Instead, use a Members[i] and Size[i] for all i. Now, we don't have to go through the whole graph to see if component[i] ==k needs to be made to k'. The complexity now reduces to O((m + n) log n)

Union find using pointers

Priority Queue

- Need two functions delete_max() to remove job with highest priority
- Insert() to insert a new job
- In an unsorted list, delete_max() requires O(n) and insert() takes O(1)
- In a sorted list, delete_max() takes O(1) and insert() takes O(n)
- Say instead of 1D array or list, we make the structure of 2D, then we see that inserting and deleting takes only O(root N). Therefore overall is O(N root N)
- Say we use trees or heaps, both the operations will take only O(log N) hence, overall O(N log N)

Heap

- Heaps have a fixed shape. The heaps needs to be filled from left to right. In case of no value, None is to be chosen.
- Max Heap states that the child needs to be less than the parent, min heap is the exact opposite
- In order to insert, you insert at the bottom first and compare as you come towards the right position.
- Insertion takes O(log N)
- To delete, remove the last proper location of the node. Replace that value in the node which was removed. Now, compare and move to right place.
- This also takes O(log N)
- Children of H[i] are H[2i + 1], H[2i +2]
- Parent of H[j] is H[floor((j-1)/2)] for j > 0
- Fresh heapifying takes O(N) time

To update values and sort

- Keep two new arrays NodetoHeap and HeaptoNode
- To get a sorted list, use heapsort. Heapify the data, delete the element one by one till the whole is empty while creating a new list to store the values in line. Delete_max takes O(log n) now, for n different numbers, overall complexity is O(n log n)

Inversions

- Inversions: an **inversion** in an array is a concept used to measure how far the array is from being sorted
- When graphed, inversions mean that there are lines crossing. If n(n-1)/2 inversions occur, that means that every pair is inverted.
- To count the number of inversions or sort, divide the list into two sorted arrays. With L and R pointers. Constantly sort and count and merge and count.
- Using merge sort, O(n log n). But brute force calculation is O(n^2)

Closest pairs

- Calculating the closest pair of points in 1 dimension takes O(n log n) while for 2 dimensional, divide the set into left and right sides and find the closest pairs among the provided pairs.
- Calculate ClosestPair(Px,Py) with the help of ClosestPair(Qx,Qy) and ClosestPair(Rx,Ry) for left and right half of P in time O(n)
- Say dq is the closest distance in Q and dr be the closest distance in R, and let d be the minimum. From the center line, just check –d and +d distances.
- Computing Px Py from P takes O(n log n). All the other actions like setting up Qx, Qy, Rx, Ry, Sy is O(n) so overall O(n log n)

Comparing DS

	Unsorted array	Sorted array	Min Heap
Min	O(n)	O(1)	O(1)
Max	O(n)	O(1)	O(n)
Insert	O(1)	O(n)	O(log n)
Delete	O(n)	O(n)	O(log n)
Pred	O(n)	O(1)	O(n)
Succ	O(n)	O(1)	O(n)

	Heap	Sorted array	Search tree
Find	O(n)	O(log n)	O(log n)
Min	O(1)	O(1)	O(log n)
Max	O(n)	O(1)	O(log n)
Insert	O(log n)	O(n)	O(log n)
Delete	O(log n)	O(n)	O(log n)
Pred	O(n)	O(1)	O(log n)
Succ	O(n)	O(1)	O(log n)

Trees

- In Binary Search trees, say the root is v, then the values in the left of the subtree < v and the values of the right of the subtree is > v. No duplicate values unlike heap where a child and a parent could also be equal.
- Minimum is the left most node in the tree, slly, rightmost node is Maximum.
- To find the successor, we check if there is a right of the subtree, if so, return that value. Else, check the parent until its parent it null or it the parent's right is not the value. Then return the value of the parent.
- The predecessor is the same check the left of the value given.
- Take a look at the video to delete any node.
- Read AVL

Notes

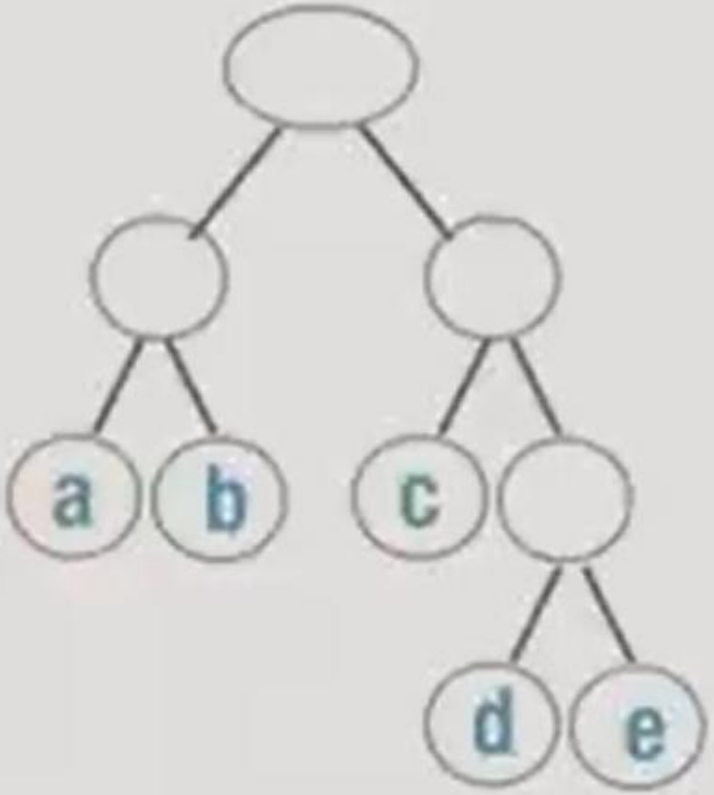
- In booking intervals:- First sort n booking by finish time, number them 1 to n. Set up and array such that ST[i] = s(i). After choosing booking j, scan ST[j+1], ST[j+2],.. Pick such that ST[k] > f(i), the overall phase is O(n log n).
- Minimizing lateness: l(j) = d(j) – f(j) (late time is the difference of deadline time – finish time)
- Inversion is when i appears before j even though the deadline of j is before i.
- There is an optimal schedule with no inversions and no idle time.
- Overall O(n log n)
- Total length of encoded messages is given by n * f(x) * | E(x) |
- Huffman code works by combining the letters and their frequencies and then splitting them up slowly in a tree.
- Extract letters with minimum frequency and replace with composite letter with combined frequency. Then, the complexity drops to O(k log k)

x	a	b	c	d	e
f(x)	0.32	0.25	0.20	0.18	0.05

Split “de” as d, e

x	a	b	c	de
f(x)	0.32	0.25	0.20	0.23

Split “cde” as c, de



x	a	b	cde
f(x)	0.32	0.25	0.43

Split “ab” as a, b

x	ab	cde
f(x)	0.57	0.43

Two letters, base case

WEEK 7

- Overlapping subproblems leads to wasteful recomputation. In order to avoid reevaluation, we create a memory table, ie, Memoization.
- Dynamic programming includes anticipating what a memory table looks like, so we can solve subproblems in a topological order. (Need to form a DAG{Directed Acyclic Graph})
- To move from one (0,0) to (m,n) u will have to make (m+n)Cn moves. Say one part is blocked. Then calculate the paths from (0,0) to point (I,j) and then from (I,j) to (m,n). Subtract the sum from original
- Paths(i,j) = paths(i-1,j) + paths(I, j -1)
  Boundary cases are:
  paths(I,0) = paths (i-1,0)
  paths(0,j) = paths(0, j -1)
  paths(0,0) = 1
- We can fill the grid with rows first or columns first or even diagonally.

WEEK 7

- The brute force approach to find the longest subword is $O(mn^2)$
- Using memorization, O(mn)
- The longest subsequence is a way that the words are present but are not attached.
- LCW(I,j) = 0 if ai = bj else, 1 + LCW(i+1,j+1)
- Same process but increment. Instead, change the values of every rows and to get the final sequence, just find the diagonals.
- ai = bj, LCS(I,j) = 1+LCS(i+1,j+1)
- If ai != bj, LCS(I,j) = max(LCS(i+1,j), LCS(i,j+1))
- EDIT DISTANCE LEFT

Matrix multiplication

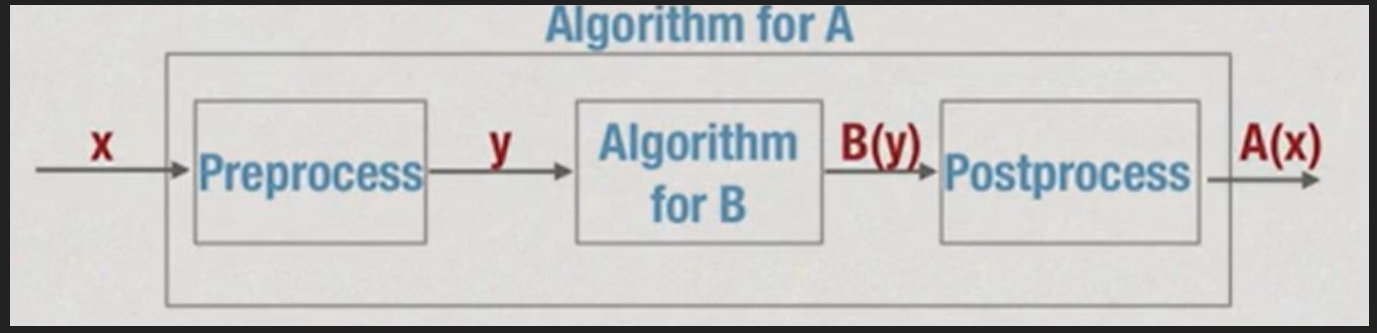
- Say we are given multiple matrices that we have to multiply and their dimensions are also given, to find the optimal way to compute the solution.
- First divide the huge problem into (M1….Mk) and from (M(k+1) … Mn) say the factor has dimensions (r,c) and (c,d) then the cost is O(r*c*d).
- Basically Cost(M1 …. Mk) + Cost(M(k+1) ….. Mn) + rcd
- As with LCS, ED, we need to fill an $O(n^2)$ size table. But filling MM[i][j] requires O(n) hence, overall complexity is $O(n^3)$

Week 8 linear programming

- Linear programming has variables in the form of ax + b = y. There are a set of constraints of linear form which are to be satisfied.
- Feasible region is convex. Meaning they are bounded within a given region.
- May be unbounded, or empty
- A 3D picture is called a POLYTOPE
- You can always construct a combination of constraints that tightly capture upper bound of an objective function.
- Create different variables to determine the final set of equations and determine the answer. In case the answer is in fractions, and you don't need it in fractions, round it to the nearest integer and check both ways and see which one gives the higher of the lower value and choose accordingly.
- Linear programming with the help of one variable per equation is not the best strategy to analyse network flows.

Week 8 Networks and flow things

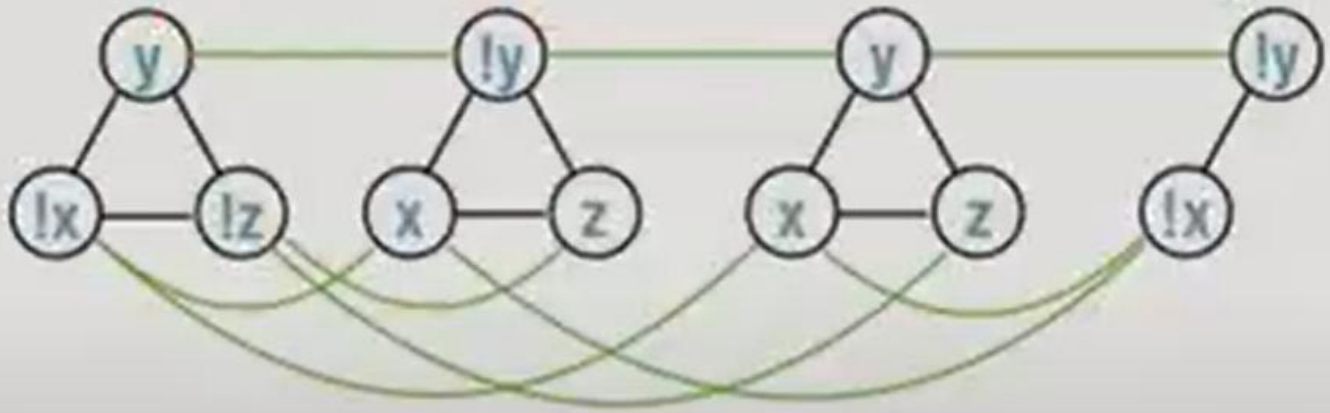
- Ford-Fulkerson Algorithm states that we should start from 0 flow then choose a non-saturated path and augment the flow as much as possible. In case we pick the wrong flow, we flow in the reverse direction.
- Find a complete path from source to sink, make the path residual (ie, make the original path 0) and then find other routes/flows, until all the vertices become 0.
- Bipartite matching is where partitioning is done into two groups in which there is no connection/vertex within one group.
- Reductions is a process used here. First, we convert the matching problem into a flow problem. From there we solve the flow problem and give it as the output of A



Reductions, checking algo and P,NP

- Clause formula C of the form (x || !y || z || …….w) disjunction of literals or variables.
- Similarly, conjunction of clauses, x & y & z & … w
- Travelling salesman problem consists of a complete graph(a node is connected to all other nodes) The city/node you start at is the city to stop at.
- (u,v) are independent if there is no edge (u,v)
- Factorization, satisfiability, travelling salesman, vertex cover, independent set.. Are all in NP.
- NP stands for non-deterministic polynomial time. Guess and check method, like trial and error.
- P is a class of problems with regular polynomial time algos. P != NP as efficient generation is better than efficient checking
- Many natural problems are in NP
- 3-SAT is when a clause has at most 3 literals.
- (v or !w or x or !y or z)
  -> (v or !w or a) and (!a or x or !y or z)
  -> (v or !w or a) and (!a or x or b) and (!b and !y and z)

(!x || y || !z) & (x || !y || z) & (x || y || z) & (!x || !y)



- Size of independent set = number of clauses
- Cook Levin theorem is where every problem in NP can be reduced to SAT.
- SAT is said to be complete for NP
- BTW, SAT stands for satisfiable problem.