Master theorem

Dividing

decreasing

$a\,T(n/b) + f(n)$ → no. of recursive calls; → Time of other steps than recurs.

$a\,T(n-b) + f(n)$ → size of each recursive call

Algorithm - Inv

Inversion Count

O(n log n)

By Merge Sort (Enhanced)

* indicates how far the array is from being sorted. inv count = 0 → if already sorted → max. reverse sorted

2	1	8	5

i=0

2	4	6

Merge step

Inversion := if for i < j a[i] > a[j]

inversion takes place

1	3	5

i=0

2	4	6

j=3

INV = 0

1

1	3	5

i=1

2	4	6

j=3

INV = 2

1	2

for i < j a[i] > a[j]

2 > 2

∴ INV = inv + mid - i

(for 3, 5)

1	3	5

i=2

2	4	6

j=5

INV = 3

1	2	4	6

12 → Strassen's Matrix Multiplication

MM (A, B, n)

if $n \leq 2$ { }

$C_{11} = P + S - T + V$

$C_{11} = R + T$

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

else

MM (A11 B11 n/2) + MM (A12 B21 n/2)

MM (A11 B11 n/2) + MM (A21 B22 n/2)

MM (A21 B12 n/2) + MM (A22 B22 n/2)

MM (A21 B12 n/2) + MM (A22 B12 n/2)

$P = (A_{11} + A_{21})(B_{12} + B_{21})$

$Q = (A_{21} + A_{22}) B_{11}$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = (A_{11} + A_{12}) B_{22}$

$U = (A_{21} - A_{11})(B_{11} + B_{12})$

$V = (A_{12} - A_{22})(B_{21} + B_{22})$

Master Method

$$T(n) \leq a\,T(n/b) + O(n^d) \quad f(n)$$

a → recursive calls; b → size of each recursive call; $O(n^d)$ → time by other steps than recursive calls

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \\ O(n^d) & \text{if } a < b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Probability Review

(1) Expectation →

$$E[X] = \sum_{i \in \text{sample space}} P(i)\,X(i)$$

Q1. Expectation of sum of 2 dice

Sample space → 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Prob. → $\frac{1}{36}, \frac{2}{36}, \frac{3}{36}, \frac{4}{36}, \frac{5}{36}, \frac{6}{36}, \frac{5}{36}, \frac{4}{36}, \frac{3}{36}, \frac{2}{36}$

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

$E[x] = \sum P_i x_i$

$= \frac{1}{36} \cdot 2 + \frac{2}{36} \cdot 3 \ldots$

$= 7$

② Conditional prob →

$P(X|Y) = \frac{P(X \cap Y)}{P(Y)}$ → Prob
→ given

Q. Given sum = 7, prob that atleast one of them has 1

$P(X|Y) = \frac{(1,6)(6,1)}{(1,6)(6,1)(2,5)(5,2)(3,4)(4,3)}$ ↳ given sum = 7

$= \frac{2}{6} = \frac{1}{3}$

③ Independent →

$P[X \cap Y] = P[X] \cdot P[Y]$

But case → n if pivot middle always
worst case →

A3. Randomized Selection Algorithm

Avg → O(n)
worst → O(nlogn)

Time → $T(n) = T(n/2) + O(n)$

$T(n) = O(n)$

Worst case when pivot is 1 elem and sorted and i is last to O(nlogn)

Rselect (a, n, ordy i)

0) if n=1 return A[1]
1) choose pivot p from A
2) partition A around p
   let j = new index of p
3) If j == i return p
4) if j > i return Rselect (1st part of A, j-1, i)
5) if j < i return Rselect (2nd part of A, n-j, i-j)

Date: / /

Cuts of A Graph

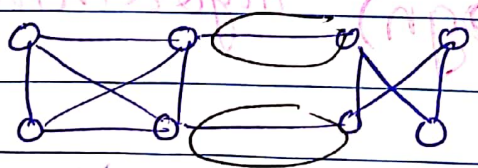
Cut → partition of vertices into two non-empty sets A & B.

2^n choices → types of cuts possible

Simple Crossing edges of a cut → edge if cut, disconnect the graph.

those edges with →
① one endpoint in each A & B
② tail in A & head in B (for directed)

Q. No. of edges crossing a min cut in the graph?



Ans-2

If removed disconnects the graph.

Parallel edges →  2 same vertices

cut → 2 vertices b/w them & edges cut graph disconnect

~~crossing edges → no. of edges b/w a cut~~

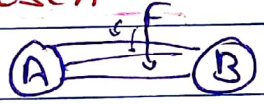
- The Minimum cut Problem → compute a cut with fewest no of crossing edges

Random Contraction Algorithm

while more than 2 vertices
- pick a edge at random
- merge u & v into a single vertex
- remove self loop

return cut represented by 2 vertices (edges connecting two vertices)

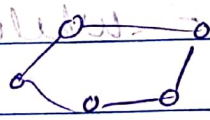
#sometimes gives the minimum cut based on edge chosen.

Failure when → 

Let min cut be AB

one of 2 the edges (A) selected cut is some point partition containing AB & won't output this cut.

success probability → $1/n^2 \approx$ or $\frac{2}{n(n-1)}$

- Min cuts → 

Lower bound → n cycle $n C_2$ n me se koi bhi 2 edges kat do.

Upper bound → contraction algo analysis → $\frac{2}{n(n-1)}$

→ No of trials to do → $\geq n C_2$ = $\frac{1}{n C_2}$

COURSE-2

Graph Search Shortest Path and Data Structures

Date: / /

(1) BFS → $O(m+n)$ using queue
→ explore nodes in layers
→ can compute shortest paths
→ can compute connected connected comp. of an undirected graph

(2) DFS → using stack
→ explore like maze, backtracking only when necessary
→ compute connected comp of directed graph

BFS Algo

BFS (graph G, start vertex s)
[all nodes initially unexplored]
- mark s as explore
- let Q = queue init with s
- while Q ≠ ∅:
  - remove first node of Q → v
  - for each edge(v,w):
    - if w unexplored
      - mark w explore
      - add w to Q

Running time → n_s → no. of nodes reachable from s
m_s → no. of edges

→ Shortest Path → will tell that shortest path from source node to v has i edges. i.e dist(v) = i

dis(v) = 0

when considering edge (v,w) Date:

if w unexplored

dis(w) = dis(v) + 1

→ only BFS can be used

→ Undirected Connectivity → O(m+n)

compute all connected components

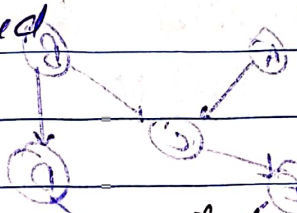
- check if network is disconnected.

Code → all node unexplored labelled to 1-n

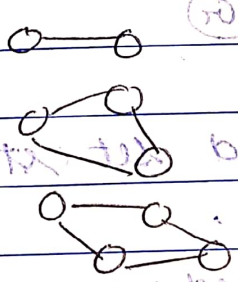
- for i = 1 to n

- if i not explored

- BFS(G, i)



*So if a graph has connected component like



so we still can compute all the nodes

3 components

Both BFS & DFS can be used But for undirected graph

BFS (O(m+n))

Stack →

DFS (graph G, start vertex s)

- mark s explored

- for every edge (s,v)

- if v unexplored

DFS (G, v)

Date: / /

DFS → topological sort → of directed graph

An ordering of a directed graph is a labelling f of G's node such that:

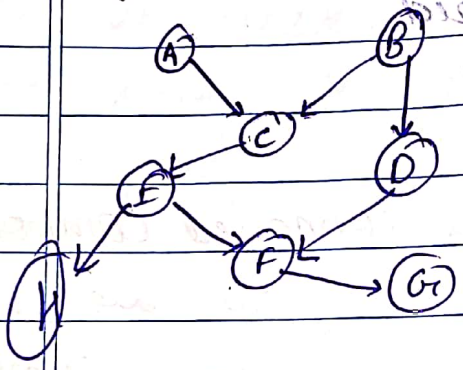
① $f(V's) \in \{1, 2, 3, \ldots, n\}$

② $(U,V) \in G \Rightarrow f(U) < f(V)$

u → v u should always appear before v.

★ not going backward ↓

As long as you don't have a cycle → definitely TS



what comes before is written before

A B C E D F G

B D A C E F G

① Put in visited set starting from chosen index.

② When explored an index, keep in the sorted set

Strongly connected components (Di

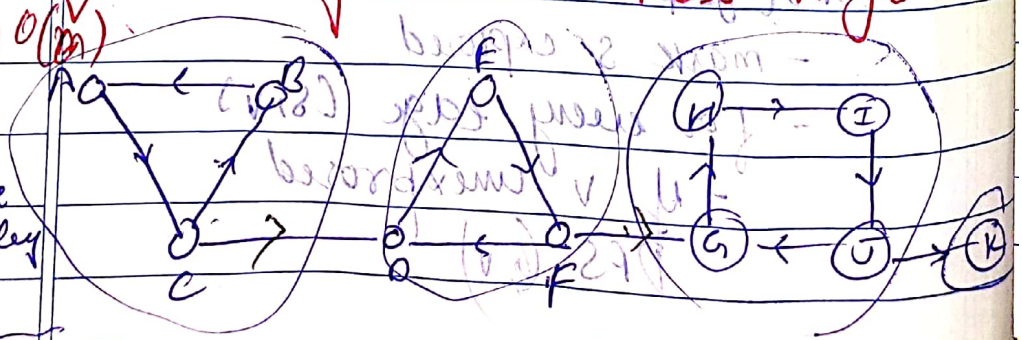
If u ⇝ v ✓

then v ⇝ u ✓

Directed graph

O(E+V) Kosaraju's Two Pass Algo →

Space → O(V)



A can be reach by B, C, B can C can

Algo →

① G is reversed
② DFS on G rev
③ DFS on G

Date: / / /

① ~~G is~~ DFS on G and make a stack on finished time taken by node
② rev G and start DFS from top node of stack,
③ Each traversal will give component

* Important

ADJACENCY LIST

① outdegree of u → Adj[u].length()

② sum of lengths of all adjacent list is |E|.

③ T(outdegree of u) = O(|Adj(u)|)
T(outdegree of all) = O(V+E)

④ Indegree of vertex (u) = no. of times it appears in all the list of adj → (E)

⑤ T(Indegree of all vertices) = O(VE)

ADJACENCY MATRIX

① O(V²) entries.
& indegree

② T(outdegree of u) = O(V)
T(" " all vertices) = O(V²)
& indegree

Week-2

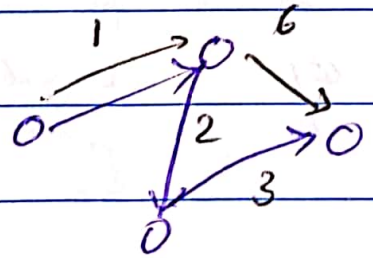
Date : / /

space O(E+V)

I ① Dijkstra's Algorithm O(E log(V))

- works in directed or undirected graph with non neg edge length.
- computes shortest path from source vertex to all other vertex

A BFS will not give shortest path when all edge length are not same.



will give →

but ans is →

for all vertex

u ← min dis vertex

~~update dist~~

for all neigh of u say i

i dist → dis[u] + u - wt i

~~if dist[i] >~~

update dis

Week - 3

I/ Heaps → a container for objects that have keys.

operation →

O(logn) ~~Insert~~ ADD → Insertion → add object

O(logn) EXTRACT → remove min/max key value

HEAPIFY → n batched inserts in O(n) time

DELETE → any element in O(log n)

Application →

① Heapsort nlogn

② Event Manager

③ Median Maintainence → everytime

a no. is inserted you tell the media

Q* Median of stream of Running Integers

Space → n Time → n log n Date: / / Heap

↓ have to do for all no.

Code: By priority queue

q_1 → max heap

q_2 → min heap.

num → no. inserted recently

```
If (q1.empty() || q1.top() > num)
    - q1.push(num)
else q2.push(num)

if (q1.size() > q2.size() + 1)
    - q2.push(q1.top())
    - q1.pop()

else if (q2.size() > q1.size() + 1)
    - q1.push(q2.top())
    - q2.pop()
```

Selecting median

The one with greatest size will give median.

```
if (q1.size() == q2.size())
    take median of max + min in
    i.e. q1.top() + q2.top()
                 2
if (q1.size() > q2.size())
    return q1.top()

if else return q2.top()
```

(4) Speeding up Dijstra's Algo

→ for naive implem →
takes O(nm) Date: / / /
→ per iteration
loop iteration finding min edge

→ for heaps
run time O(nlogn)

→ Balanced Binary Search Tree.

Rank → # of keys less than / equal to a given value.

OPERATION	RUNNING TIME
search	O(logn)
Select	O(logn)
min / max	O(logn)
Pred / succ	O(logn)
Rank	O(logn)
output	O(n)
Insert	O(logn)
delete	O(logn)

* If want have static data set + don't want del / insert → use sorted array
* If insertion / del / track of smallest ele → use Heap
* If dont want min, max, ordering, just want to know if that data exists in it / not → Hash Table

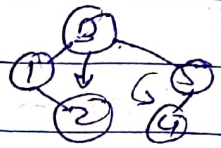
★ Want all these operation in log time → Balanced Binary Search Tree

Date: / /

BST

$\log_2 n \leq$ Height $\leq n$

★ In any BST, the worst case running time of search (or insert) operation containing n keys is
→ O(height)



★ To compute pred of key k:

① EASY CASE → k's left subtree is non-empty
return max key in left subtree

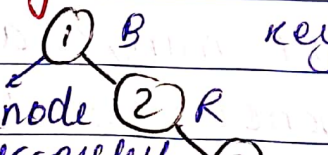
② ELSE → follow parent pointer until you get to key < k, i.e. happens 1st time you turn left

• Red Black Tree

popular Balanced Tree

- each node is red / black.
- root is black.
- no two reds in a row
  red node → black children
- every root → null path has same no. of black nodes.

★ A chain of length 3 cannot be red-black



of an unsuccessful search of 0

Date: / / /

Week-4

Hash Table * Use best for Lookups.

Purpose → Maintain a set of stuff.

Operation →

INSERTION → O(1)
DELETE → O(1)
LOOK UP → O(1)

} using a key

Application →

(1) DEDUPLICATION → removing duplicates

Sol → when new object x arrives
- look up x in hash table H
- if not found insert x in H

(2) The 2-SUM PROBLEM →

Input → unsorted array of n int, Target sum t

check if 2 int sum = t.

Sol

with DS (n) Hash Table
① Insert element of A in H n × O(1) = O(n)
② for each x in A look up t-x in H.
n × (O(1)) = O(n)

without DS (nlogn)
① Sort A (nlogn)
② binary search for x in A look for t-x in A.
(nlogn)

③ symbol tables in compilers

④ block black hackers IP addresses

Date: /

Load factor of HT →

$\alpha := \frac{\#\text{ of objects in HT}}{\#\text{ of buckets in HT}}$

- should be O(1) for operations to run in constant time or in changing not much ?!

Should have good hash fn

Bloom Filters

Fast inserts & lookup

Pros	Cons
① More space efficient	① can't store objects
②	② no deletion
	③ small false pos. probability

Applications →

① spell checker → insert in BF check for the spelling if ex
② list of forbidden passwords.
→ ③ network routers

Ingredients →

① array of n bits (so n = # of bits per object (in data set))

② k hash functions $(h_1, h_2 \dots h_k)$

Insert (x) → for $i = 1, 2, 3 \dots k$
set $A[h_i(x)] = 1$

Lookup (x) : → Return True
⇒ $A[h_i(x) = 1]$ for $i = 1, 2, 3 \dots k$

* No false neg [if x inserted, lookup ✓]
* false +ve if all $k(h_i(x)$'s) already set to 1 by other insertions

Probability of false +ve →

$$\epsilon = \left(1 - e^{-k/b}\right)^k$$

$b \rightarrow$ no. of bits/object
$k \rightarrow$ no. of Hash f^n
$= 0.693 \times b$

Course-3

Date: / /

Min Spanning Tree

Spanning Tree
A graph which contains all vertices with min no. of edges.
- A spanning subgraph
- a tree (no cycle)

MST → Total of weights of edges should be min.
→ It has n-1 edges.

Measuring similarity b/w 2 strings by sequence alignment

AGGGCT → AGGGCT
AGGCA → AGG-CA
gap mismatch

→ Straight sol → DP.

Greedy Algo

Date:

~~① Exact Knapsack~~
① Dijkstra's Algo
② Kruskal's MST
③ Prim's MST
④ Huffman coding
⑤ Activity selection Prob
⑥ Job sequence

use when
① global optimum can be arrived at by selecting local opt
② An opt soln contains an opt soln to its sub problem

An algo paradigm that builds up a soln piece by piece, always choosing the next piece that offers the most obvious & immediate benefit.

sorted input → O(n)
unsorted O(nlogn)

eg ①

Activity selection Prob.

you are given n activities with their finishing & end times. Select the max activities a person can perform assuming at a time one activity can only be done.

- sort act acc to finish time
- select 1st activity in sorted list. print
- Do - if start time of activity ≥ finish time of prev
  - print

Disjoint sets

Date: / /

A ds that keeps track of set of elements partitioned into a no. of disjoint subsets.

Union Find Algo → O(n)

→ Find → determine which subset a particular element is in.

→ Union → join two subsets into single subset

To check whether undirected graph has cycle

(* memset (pointer to array, ch to copy, size))

memset (parent, -1, sizeof (V))

- For every edge from u to v
  - x = find (u)
    y = find (v)
    if (x == y)
      return 1
    Union (x, y)

find (i) O(n)

```
if (parent[i] == -1)
    return i
return (parent[i])
```

union (x, y) O(n)

```
xs = find (x)
ys = find (y)
parent[xs] = ys;
```

Optimised version to detect cycle — Find by path compression O(log V) — union by Rank O(log V)

NOTE →
- while union of 2 sets
- if node has same Rank : Rank of new absolute parent increases by 1
- Having diff Rank : Rank doesn't change

find(i)
if(i.parent != i)
  i.parent = find(i.parent)
return i.parent

union
x_s = find(x)
y_s = find(y)
if(x_s rank < y_s rank)
  x_s.parent = y_s
else if ——— >
  y_s.par = x_s
  x_s.rank++

O(ElogE) or O(ElogV)

Kruskal MST Algorithm

connected undirected graph

For finding MST — MST will have → V-1 edges.

- Sort all edges in non decreasing order of their weights
- pick smallest edge. Check if it forms cycle with spanning tree formed so far.
- If cycle not — include edge
  - else discard
- Repeat until there V-1 edges in spanning tree.

TC → O(E log E + E log V) ≈ O(E log E) or O(E log V)
(sort, find, union)

Prims MST for Matrices → $O(V^2)$

Date: / /

- A set mstSet = track of vertices included in MST
- Assign key value to all vertices ∞.
  key[0] = 0
- while MST doesn't include all vertices -
  - pick u → min + mst = false
  - mst[u] = true
  - update adjacent of u if for each v
    wt[u→v] < key[v]
    ✓ - ~~set~~ key[v] = wt[u→v]
    - parent[v] = u

∴ $O(E \log V)$

Prims MST for Adjacency List

- Create min heap of V | key , key[0] = 0, key[ ] = ∞
- while minheap ! 0
  - Extract → u
  - for adjacent vtex of u - v
    - if (v is in heap + key[v] > wt[u→
      - key[v] = wt[u→v]

Time $O(E+V) * O(\log V)$

inner loop ↓ decrease key ( )

Huffman Coding → n log n (for n inputs) → extract

Date: / /

Efficient method of compressing data without losing info.

Creating Huffman Tree

→ Input → array of ~~unique~~ char + freq

→ min heap with free node = char freq

→ Extract two node

Create internal node with freq = sum of 2 freq

first extracted node = left

second = right child

Add to heap.

→ Till one node left repeat

For sorted input → O(n)

1→ Create two queues q_1, q_2

2→ push all nodes in increasing order in q_1

3→ Deque two min node by →

- if q_2.empty() — deque q_1.front()
- if q_1.empty() — deq q_2.front()
- compare q_1.f & q_2.front choose min

4) Create root attach left, right push in q_2

5 Repeat 3, 4 till q_1 is empty & q_2 is 1 node.