PDSA

Week-1

L1.1 Intro. to Jupyter NB & Google Colab

- manual - text editor to write code
  - run at the command line
- IDE - single app. to write and run code
  - quick update - run cycle
  - debugging, testing, Eg. replit
- share your code → collaborative dev., export your results
- documentation - interleave with the code
- switching btw. diff. versions of code, export and import your progs., preserve your output
- Jupyter NB
  - a seq. of cells, 1-D spreadsheet
  - cells hold raw text, markdown notation for formatting (www.markdownguide.org)
  - Edit and re-run indi. cells to update environment
  - supports diff. kernels → Julia, Python, R
  - widely used to document and disseminate ML projects (kaggle)
  - ACM software sys. award 2017
- google colab → free to use
  ↳ colab.research.google.com
  - customized jupyter nb
  - all std. packages of ML are preloaded - scikit-learn, tensorflow
  - access to GPU hardware

L1.2 Implementation of Python Codes (Part-1)

- # bigger heading, ## smaller heading,
  - list → gives list ⇒ markdown lang. for text documentation

Eg.
```
def fac(n)
    fl = []
    for i in range(1, n+1):
        if n % i == 0
            fl.append(i)
    return(fl)
```

```
def prime(n)
    return(len(fac(n)) == 2)
```

```
pl = []
for i in range(1,101):
    if (prime(i)) =
        pl.append(i)
print(pl)
```

L1.3 Python Recap-1

- gcd(m, n) / hcf
  - ∴ largest k that divides both m & n
  - gcd(8, 12) = 4 & gcd(18, 25) = 1
- it always exists, if nothing then 1
- gcd(m, n) ≤ min(m, n)

```
def gcd(m, n):
    cf = []
    for i in range(1, min(m,n)+1):
        if (m%i == 0 and (n%i == 0:
            cf.append(i)
    return(cf[-1])
```

Otherwise last factor won't be taken

indicates list

control flow

⇒ after if —

```
    mcf = i
return(mcf)
```

L1.4 Python Recap-2

- checking primality ⇒ counting primes
- list the first m primes

```
def fp(m):
    (count, i, pl) = (0, 1, [])
    while (count < m):
        if prime(i):
            (count, pl) = (count+1, pl+[i])
        i = i+1
return(pl)
```

- directly check if n has a factor btw 2

```
def fp(n):
    x = True
    for i in range(2, n):
        if (n%i) = 0
            x = False
            break # about loop
    return x
```

- Properties of primes —

(i) ∞ primes

(ii) twin primes : p, p+2 ⇒ ∞ twin p

(iii) prime diff.

```
def pd(n):
    lp = 2
    pdd = {}
    for i in range(3, n+1):
        if prime(i)
            d = i - lp
            lp = i
            if d in pdd.keys():
                pdd[d] = pdd[d] + 1
            else:
                pdd[d] = 1
    return(pdd)
```

L1.5 Python Recap-3

- d divides m & n ⇒ m = ad, n = bd

  (m − n) = (a − b) d

⇒ d also divides (m − n)

```
def gcd(m,n)
    (a,b) = (max(m,n), min(m,n))
    if a%b == 0:
        return b        # base case
    else:
        return(gcd(b, a-b))
```

- Euclid's algorithm

same as before

else:
return (gcd (b, a%b))

- It is one of the first non-trivial algorithm.

L1.6 Exception handling

- Common errors (i) y = x/z, z = 0
  (ii) y = int(s) not as int
  (iii) y = 5*x, x not defined
  (iv) read a file, file doesn't exist
- recover gracefully, using Exceptions handling
- syntax error → u can't correct-lines
- correct ver possible in name error, zero div. error, index error
- unhandled exceptions aborts execution

handling except:

```
try:
    → code
except IndexError:
    → handle index error
except:
    → handle other exceptions
else:
    → execute if try runs w/o errors
```

- we can use exceptions freely, as in a dictionary

```
if b in scores.keys():
    scores[b].append(s)
else:
    scores[b] = [s]
```

```
try:
    scores[b].append(s)
except KeyError:
    scores[b] = [s]
```

L1.7 Classes & Objects

- Class — template for a data type, how data is stored, how public func^ns manipulate data.
- obj. - concrete instance of template

```
class Point:
    def __init__(self, a=0, b=0):        # initialize int, real x, y
        self.x = a
        self.y = b                       # 1st parameter is self

    def translate(self, delx, dely):     # translation shift form Δx, Δy
        self.x += delx
        self.y += dely

    def od(self):
        import math
        d = math.sqrt(self.x*self.x + ...)
        return d
```

- Polar coordinates (a, θ) instead of (x, y)
  θ = tan⁻¹(y/x)

```
import math
class point:
    def __init__(self):                  # a=0, b=0
        self.a = math.sqrt(a*a + b*b)
        if a==0:
            self.theta = math.pi/
        else:
            " = math.atan(b/
```

```
def od(self):
    return (self.a)
```

but translation becomes more elongated.

- __init__() — constructor
- __str__() — convert obj. to string
- __add__() — invoked by +
- __mult__()
- __lt__()
- __gt__()

*
<
Δ = 1

L1.8 Implementaⁿ of Python Codes (Part-2)

- all codes in google colab –
  → p = Point(3,4) q = Point(7,10)
  → p.od(), q.od()

L1.9 Timing Our Code

- Library time → perf_time()
  - 2 consecutive readings gives the int.
  - default unit is seconds.

```
import time
start = time.perf_counter()
# execute code
end = time.perf_counter()
elapsed = start - time
```

```
import time ⇒
class Timer:
    def __init__(self):
        self._start_time = 0
        self._elapsed_time = 0
    def start(self):
        self._start_time = time.perf_counter()
    def stop(self):
        self._elapsed_time = time.perf_counter()
                             - self.start_time
    def elapsed(self):
        return(self._elapsed_time)
```

L1.10 Implementaⁿ of Python Codes (Part-3)

- refer to code
- create a error - Time error
  raise 'an excepⁿ'

- trivial loop to see Python preform 10^7 operaⁿˢ in 1 second

L1.11 Why Efficiency Matters?

- SIM cards linked to Aadhar Card
- Validaⁿ of Aadhar details to SIM card
- simple nested loop for matching the 2 (10^9)
  ↳ no. of sim cards $\times$ no. of aadhar card (10^9)
  10^{18} times of nested loop

10^7 in 1s

1 — $\frac{1}{10^7}$ ⇒ 10^{11} seconds
= 3200 years

- propose a data (bday) → interval of (yes, earlier, later) possibilities ⇒ query the mp. → halves the interval
  365 → 182 → 91 → 45 → 22 → 11 → 5 → 2 → 1
- assume Aadhar details are sorted out as Aadhar number ⇒ use the halving strategy to check each SIM card
  ↳ 30 queries $\times$ 10^9 = 3000 sec. = 50 min
- arranging the data results in a much more efficient solution
- both algorithms and data structures matter

L1.12 Implementaⁿ of Python Codes (Part-4)

- naive algo's stop work
- recursion error → limit of remembering answer

CLASSTIME | Page No. | Date

Week-2

L2.1 Analysis of Algorithms

- naive approach takes 1000s of yrs, smarter sol^n takes a few minutes
- running time and space
  depends on processing-time (indep. of hardware) and storage is, ltd. by available memory
- time depends on input size.
  input size $\Rightarrow n$
  run time $\rightarrow t(n)$
- diff. inputs of size n take diff. amt. of time
  Eg. $10^4 \Rightarrow n$ Naive $\Rightarrow t(n) \approx n^2$
  (sim card eg.) Clever $\Rightarrow t(n) \approx n \log_2 n$
  Eg. find closest pair of objs.
  naive : $\rightarrow n^2$
  clever : $\rightarrow n \log_2 n$

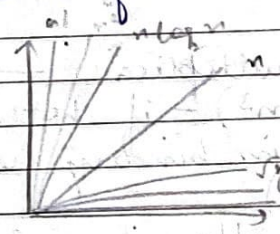
- when comparing $t(n)$, focus on orders of magnitude, ignore constant factors
- asymptotic complexity — what happens in the limit, as n becomes large
- typical growth func^n — log, poly., expon.
- analysis should be independent of the underlying hardware → don't use actual time, measure in basic op^n (comparisons, assignment)
- typically a natural parameter → size of a list, no. of objects, no. of vertices and edges in this graph
- numeric problems? is n a prime ⇒ no. of digits is a natural measure of input size

CLASSTIME | Page No. | Date

- performance varies across input instances, take the 'avg.' behaviour ⇒ but this isn't practical ∴, we take 'worst case' input ⇒ upper bound for worst case guarantees good performance

L2.2 Comparing Orders of Magnitude

- $f(n) = n^2$ grows faster than $g(n) = 5000n$
- $f(n)$ is said to be $O(g(n))$, if we can find constants c and x_0 such that $c \cdot g(x)$ is an upper bound for $f(x)$ for x beyond x_0.
  $f(x) \leq c\,g(x)$ for every $x \geq x_0$



Eg. $100n + 5$ is $O(n^2)$
$\rightarrow 100n + 5 \leq 100n + n = 101n$, for $n \geq 5$
$\Rightarrow 101n \leq 101n^2$
$\rightarrow n_0 = 5$, $c = 101$ or $n_0 = 1$, $c = 105$

Eg. $100n^2 + 20n + 5$ is $O(n^2)$ but n^3 is not $O(n^2)$

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$, then $[f_1(n) + f_2(n)]$ is $O(\max(g_1(n), g_2(n))$
  ↳ least efficient phase is the upper bound for the whole algorithm

- $f(x)$ is said to $\Omega(g(x))$, such that $cg(x)$ is a lower bound for $f(x)$ for x beyond x_0
  $f(x) \geq cg(x)$ $\forall$ $x \geq x_0$ Eg. n^3 is Ωn^2

- gen., we establish lower bounds for a problem rather than an indi. algo.
- $f(x)$ is said to $O(g(x))$ if it is both $O(g(x))$ and $\Omega(g(x)) \Rightarrow c_1 g(x) \leq f(x) \leq c_2 g(x)$ $x \geq x_0$

Eg. $\frac{n(n-1)}{2}$ is $O(n^2)$

L2.3 Calculating Complexity — Egs.

- iterative and recursive prog.
- max. element in a list — $O(n)$
- check whether a list contains duplicates
  - worst case $i \Rightarrow len(L)$, $j \Rightarrow len(L)$ no duplicates $O(n^2)$
- matrix multiplicⁿ → 3 nested loops $O(n^3)$ — if both are $n \times n$
- no. of bits in binary representaⁿ of n — $O(\log n)$
- Towers of Hanoi → 3 pegs A, B, C
  - recurrence

$$M(n) = 2M(n-1) + 1$$
$$= 2^2 M(n-2) + (2+1) \ldots$$
$$= 2^k M(n-k) + (2^k - 1))$$
$$\ldots = 2^n - 1 \Rightarrow \text{exponential}$$

- iterative → focus on loops
- recursive → write and solve a recurrence

L2.4 Searching in List

- is value v present in list L?
- naive solⁿ → scans the list, input size n, length of list ⇒ worst case complexity is $O(n)$
- what if L is sorted in asc. order, comparision of v with the mp. of L, concept of binary search → $O(\log n)$
- $T(n)$: the time to search a list of length n

$$\left.\begin{array}{ll} n = 0 & T(n) = 1 \\ n > 0 & T(n) = T(n//2) + 1 \end{array}\right\} \text{Recurrence}$$

⇒ solve by unwinding

$$T(n) = T(n//2) + 1$$
$$= (T(n//4) + 1) + 1 = T(n//2^2) + 2$$
$$= T(n//2^k) + \underbrace{1 + \ldots + 1}_{k}$$
$$= T(1) + k \quad \text{for } k = \log n = 2 + \log n$$
$$\Rightarrow O(\log n)$$

L2.5 Selection Sort

- sorting list helps in ⇒ binary search, finding median, checking duplicates, building freq. table of values
- Eg. TA for course
- select the next element in sorted order, but try to avoid using the second list ⇒ eventually the list is arranged in place in ascending order
- correctness follows from the invariant
- efficiency $T(n) \rightarrow O(n^2)$
- this is intuitive algo. to sort a list, every case is worst case complexity itself with same time every time.

L2.6 Insertion Sort

- same eg.
- move the first paper, second paper check whether higher or lower, and so on, insert each paper
- start building a new sorted list ⇒ pick next element and insert it into the sorted list ⇒ an iterative formulation
  - ↳ assume L[:i] is sorted
    insert L[i] in L[:i]
- recursive way → inductively sort L[:i]
  insert L[i] in L[:i]
- correctness follows from the invariant
- efficiency → T(n) is $O(n^2)$
- in case of recursive method,
  $T(n) \rightarrow$ insert $T_S(n) \rightarrow$ Isort
  $T(n)$ is $O(n^2)$
- cards intuition → we have to create a sorted list
- worst case is $O(n^2)$ but the gen. time taken is $O(n)$

L2.7 Merge Sort

- to bring complexity below $O(n^2)$.
- divide the list into 2 halves
- combine 2 sorted lists A & B into a single sorted list C ⇒ merging A & B
- Let n be the length of L ⇒ now do the same halving thing recursively till only 1 element left
- divide and conquer algorithm
  break — solve — combine the soln
- if len(A) ≤ 1, nothing to done otherwise ⇒ sort and merge
- this was done to reduce the complexity of $O(n^2)$ of selection and insertion sort

L2.8 Analysis of Merge Sort

- merge A of length m, B of length n
  - ↳ output list C has length (m+n)
  - ↳ in each iteration we add at least 1 element to C
  - → merge → $O(m+n) \xrightarrow{(m=n)} O(n)$
- in recurrence,
  $T(n) = 2^k T(n/2^k) + kn$
- $O(n \log n)$ → effective ⇒ best time possible in sorting
- recursions on merge are possible
- merge always creates a new list to hold the merged elements, this and recursive calls and returns are expensive

L2.9 Implementation of Searching & Sorting Algo.

- set up Timer class to time executions
- naive search & binary search
  10.6 s 0.83 s
  ↳ for looking for a odd no. in 10^5 nos.
- selection sort → 1.26 sec.
- insertion sort → on already sorted, performance is good (0.008 sec) but when not sorted even bad than selection sort (2 sec.

CLASSTIME | Page No.
Date / /

Week-3

L3.1 Quick Sort

- 1 new list and costly new storage and recursive calls and returns are expensive in merge sort
- divide and conquer w/o merging
  $T(n) = 2T(n/2) + n \Rightarrow T(n) = O(n)$
  $\therefore T(n) = O(n \log n)$
- how to find the median? Instead pick some element in L — Pivot by (C.A.R. Hoare)
  ↳ partition L into lower and upper parts w.r.t to pivot
- partitioning is impt. — Pivot, Lower, Upper, Unclassified
- in this we avoid a merge step ⇒ allows one in place sort

L3.2 Analysis of Quick Sort

- partitioning wrt. pivot ⇒ $O(n)$
  If the pivot is median ⇒ $O(n \log n)$
- Worst case? Pivot is max or min → $O(n^2)$
  ↳ already sorted array: worst case!
- However, the avg. case is $O(n \log n)$
- any fixed choice of pivot allows us to construct worst case input, instead, choose pivot position randomly at each step
  ↳ expected running time ⇒ $O(n \log n)$
- recursive calls, in practice, quicksort is very fast → excel, pglrun

L3.3 Implementation of Quick Sort algo.

- merge sort → 10^6 input ⇒ 9 sec.
  quick sort → 10^6 input ⇒ 7 sec. } random

L3.4 Concluding remarks on sorting algos.

- stable sorting — often list values are tuples, sometimes we have to sort out things which are already sorted → this is crucial in many app.
- quick sort is not stable (swapping values disturbs the order)
- merge sort is stable if we merge carefully
- minimizing the data movement
- merge sort is used for 'external' sorting
- heapsort — $O(n \log n)$
- or use hybrid - structure, used divide and conquer till n is very large, later change the strategy

L3.5 Diff. btw. Lists & Arrays

- storing a seq. of values — List, Array.
- List
  - Flexible length, easy to modify the items, values are scattered in memory
- Array
  - fixed size, allocate a block of memory, supports random access
- List → seq of nodes, 'linked' list, easy to modify, insertion and del^n is easy via local 'plumbing', flexible size, takes time $O(i)$
- Arrays → fixed size, contiguous block of memory, random access, accessing any i takes same time, inserting and deleting elements is expensive
  - exchange A[i] & A[j] → constant - array
    → O(n) - lists
  - delete A[i], insert v → constant - list
    → O(n) - arrays
  - need to keep implementa^n in mind while analysing data struc.
  - binary search work well with array, but insertion is problem with array
  - either way — O(n)
  - how does it work in Python?

L3.6 Designing a Flexible List and Op^{ns} on the same

- Python class Node → list is a seq. of nodes
  → self.value is the stored value
  → self.next pts. to next node
- L_1 = node() → empty list
  node(5) → singleton list
- appending to a list
- insert at the start of the list
- delete a value v
- use a linked list of nods to implement a flexible list

L3.7 Implementa^n of Lists in Python

- accessing an element in lists → O(n)
- python lists are not implemented as flexible linked lists → it maps to the array and assign a fixed block for the list
- L.append() and L.pop() — O(1)
  insertion & deletion — O(n)

- effectively, Python lists behave more like arrays
- arrays are useful representing matrices

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ [[0,1],[1,0]]

- zlist = [0,0,0]
  zmatrix = [zlist, zlist, zlist]
  mutability aliases different values
  zmatrix = [[0 for i in range(3)] for j in range(3)]

  Or

  import numpy as np
  zmatrix = np.zeros(shape = (3,3))
- arrange is eg. of range of lists
- numpy arrays are easier to use

L3.8 Implementation of Dict. in Python

- array/list allows access-them positional indices
- dict. allows access them arbitrary keys, access time is same for all keys
- underlying storage is an array
- Hash func^n → h:S → X, maps a set of values S to small range of integers
  → |X| << |S|, collisions h(s) = h(s'), s ≠ s'
  → a good 'hash func^n' will minimize collisions
  → SHA-256 → range is 256 bits
  ↳ avoid uploading duplicate for cloud storage
- Hash-table — An array of size n combined with h
- deal with collisions
  1. open addressing (closed hashing)
     ↳ probe a seq. of alternative slots in the same array
  2. open hashing
     ↳ each slot in the array pts. to a list of values, insert into the list for that given slot
- diction. key must be immutable; if value Δs, hash also Δs
- many heuristics / optimizations possible for dealings

L3.9 Implementation of diff. btw. List & arrays

- create a empty list of 10^4 appends → 1.549 sec.
- create a empty list, 10^5 inserts (0,i) → 2.4
- array searching — in numpy
  naive search — takes too long time
  binary search — works much better
- but other sorting algo on arrays takes much longer time on indexing

Week-4

L4.1 Introduct^n to Graphs

- visualising rela^ns as graphs $A \subseteq T \times C$
  $A = \{(t,c) \mid (t,c) \in T \times C, t \text{ teaches } c\}$
- Graphs : $G = (V, E)$ → set of edges
  ↓ is a set of vertices
- Directed graph ⇒ $(u,v) \in E \not\Rightarrow (v,u) \in E$
- Undirected graph ⇒ $(u,v) \in E \Rightarrow (v,u) \in E$
- A path is a seq. of vertices $v_1, v_2 \cdots v_k$ connected by edgs. It does not visit a vertex twice. A seq. that revisits a vertex is c/d a walk.
- Paths in directed graphs. Vertex is reachable from vertex u, if there is a path $u \to v$. Shortest? Connected graph?
- Map colouring — states neighbours should have a different colour, how many colours needed? $(u,v) \in E \Rightarrow c(u) \neq c(v)$
- 4 colour theorm — for planar graphs from geographical maps, 4 colours suffice
- vertex cover — min. no. of cameras needed so that all corridors are covered. V → intersec^n, E — corridor segments
- matching ; class project with buds
- A ~~parent~~ B (directed), A ~~friend~~ B (undirected)

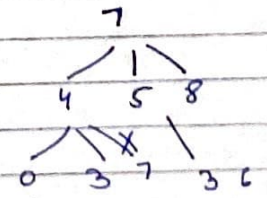
L4.2 Representing Graphs

- adjacency matrix $A[i,j] = 1$, if $(i,j) \in E$
- edges = [.. -list]
  import numpy as np
  A = np.0s (shape = (10,10))
  for (i,j) in edges
  A[i,j] = 1
- undirected graph is symmetric across main diagonal
- neighbours of i → column j with entry 1, scan row i to identify neighbours of i
- degree of vertex i, for directed graphs, outdegree and indegree
- adjacency matrix → size is n^2
  undirected : $|E| \leq \frac{n(n-1)}{2}$

  directed : $|E| \leq n(n-1)$
- adjacency list — list of neighbours for each vertex
  A list = {} └ eq. less space
  for i in range(10):
  Alist [i] = []
  for (i,j) in edges:
  Alist [i] . append (j)
- choose represent^n depending on requirem^

L4.3 BFS

- explore the graph wrt. by wrt., each vertex vertex has to be explored
- initially visited (u) = False
- A queue — FIFO (first in, first out)



7 → 4 → 5 → 8 → 0 →

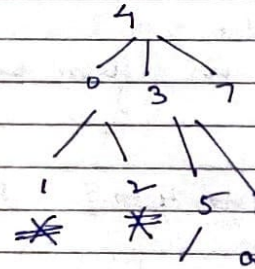
BFS

going like this

- BFS for adj. matrix — $O(n^2)$
- BFS for adj. list — $O(n+m)$
  - eg. of amortized analysis
- if $m << n^2$, working with adj. lists is much more efficient, for graphs $O(n+m)$ is best possible complexity
- recover path from i to k? from k, follow parent links to track back a path to i.
- by recording the lvl. at which a vertex is visited, we get its distance from the source vertex
- BFS is strategy to explore a graph, lvl. by lvl.
- parent info. + lvl. info. → edges are also labelled with a cost, this is known as weighted graphs.

L4.4 DFS

- stack — LIFO (last in, first out) (most recently suspended is checked first



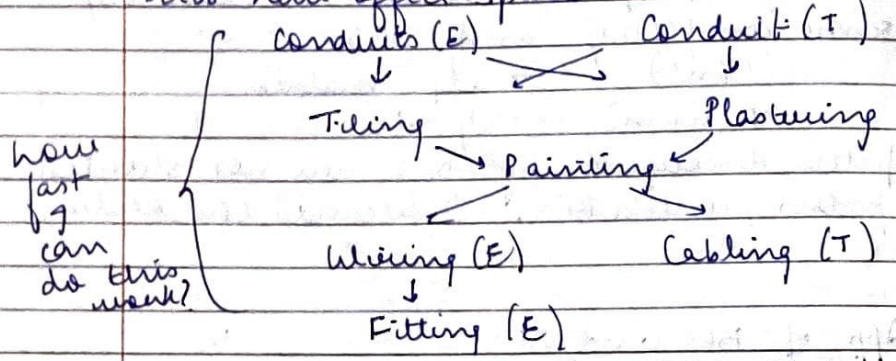
- DFS is most natural to implement recursively for each unvisited neighbour of v, call DFS(v)
- no need to maintain a stack, make visited and parent global, recursion maintains stack, first, separate the initialization step
- same complexity as of BFS
  - $O(n^2)$ — adj. matrix
  - $O(n+m)$ — adj. list
- paths discovered by DFS are not shortest paths, unlike BFS. It records the order.

L4.5 App. of BFS and DFS

- undirected graph is connected if every vertex is reachable from any other vertex
- in a disconnected graph, we can identify the connected components → use the component number strategy
- a cycle is is path, that starts and end at the same vertex, it shouldn't repeat edges.
- a graph is acyclic, if it has no cycles
- edges explored by BFS form a tree / tree per component, collec^n of trees is forest
- in a directed graph, a cycle must follow the same direc^n
- diff. type of non-tree edges — Forward, Back, Cross edges
  - Only back corresponds to cycles
- Tree edge / forward edge (u, v)
  Interval [pre(u), post(u)] contains [pre(v), post(v)]
- Back edge (u, v)
  Interval [pre(v), post(v)] contains [pre(u), post(u)]
- Cross edge (u, v) → intervals are disjoint

L4.6 Intro. to DAGs

- tasks and dependencies Eg. startup moving into new office spaces



- we have to find a schedule → Topological Sorting, longest paths

L4.7 Topological Sorting

- enumerate $V = \{0, \ldots, i\}$

Path $i \rightsquigarrow j$, i must be listed before j

claim: every DAG can be topologically sorted

- start with vertices and no dependencies, then list vertices whose dependancy has already been listed

claim: every DAG has a vertex with indegree 0

- topological sort algo.
  1. compute indegree of each vertex
  2. list a vertex (0 indegree) and remove it from DAG
  3. update indegrees
  4. find another vertex with indegree 0
  5. repeat till all vertices are listed
- initializing degrees is $O(n^2)$, loop n times
- overall, $O(n^2)$
- using adj. lists — $O(m+n)$
- always, more than 1 topological sort is possible, it depends on choice of which next vertex

L4.8 Longest paths in DAGs

- Topological sort gives feasible schedule
- Eg. DAGs of precedence of courses
- longest path
- if indegree $(i) = 0$, longest-path-to $(i) = 0$
- longest-path-to $(i) = 1 + \max \{\text{longest-path-to}(j) \mid (j, i) \in E\}$
- longest path algo.
  1. compute indegree of each vertex
  2. initialize to 0 (longest path)
  3. list a vertex with indegree 0 and remove it
  4. update indegrees, longest path
  5. repeat till all vertices are visited
- we use adj. lists for the same → $O(m+n)$
- longest path makes sense for cyclic graphs (path has at most $(n-1)$ edges)
- computing longest paths in arbitrary graphs is much harder than for DAGs — no good strategy is present

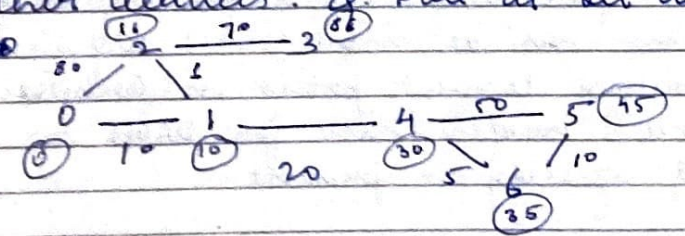
Week-5

L5.1 Shortest Paths in weighted graphs

- BFS computes shortest path, in terms of no. of ~~edges~~
- may assign value to edges → weighted graphs
- adj. matrix → weight is always 0 if no edge
- adj list → records weight with edge info.
- weighted shortest path need not have least no. of steps
- single source shortest paths — find shortest paths from a vertex to every other vertex
- all pairs shortest paths — find shortest paths btw. every pair of vertices
- -ve edge weights. Eg. Taxi driver going home at the end of day
- -ve cycles → the concept of shortest path doesn't exist

L5.2 Single Source Shortest Path (Dijkshtra's Algo)

- computes shortest paths from 0 to all other vertices. Eg. Fire at oil depot 0



- compute exp. burn-time for each vertex
  ↳ Edsger W Dijkstra's Algo.
- each new shortest path we discover extends an earlier one — Greedy strategy
- by induc", assume we have found shortest path to all vertices already burnt
- we can not use Dijkstra's algo for -ve edge weights
- maintain 2 dictionaries with vertices as keys
  → visited, initially false for all v
  → distance, initially ∞ for all v
- ∞ takes $O(n^2)$ time, main loop runs n times → Overall $O(n^2)$ → adj. matrix
  ↳ adj. list too

L5.3 Single Source Shortest Path with -ve wts (Bellman Ford Algo.)

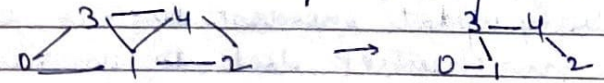
- in prev algo., correctness req. non-ve edge weights only
- shortest route to every vertex is a path, no loops
- Min. weight path for $0 \to k$ is $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \dots \xrightarrow{w_{t-1}} j_{t-1} \xrightarrow{w_t} k$
- repeatedly update shortest dist. to each vertex based on shortest dist. to its neighbours
- Initialize (source vertex 0)
  D(j): min dist. till vertex j
  $D(i) = \begin{cases} 0, & \text{if } i = j = 0 \\ \infty, & \text{otherwise} \end{cases}$
- works for directed and undirected graphs
- if there was a -ve cycle, dist would continue to decrease
- adj. matrix takes $O(n^3)$
- adj. list takes $O(mn)$
- if this algo. does not converge after $(n-1)$ iterations, there is a -ve cycle.

L5.4 All pairs shortest paths (Floyd-Warshall Algo.)

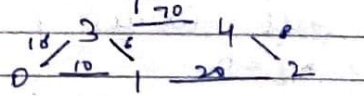
- transitive closure, $A^+ = A + A^2 + \cdots + A^{n-1}$
- $B^k[i,j] = 1$ if there is path from $i \to j$ via vertices $\{0, 1, \ldots, k-1\}$
- Warshall's algo. — also computes transitive closure
- SP^0 & SP^1 are same because 0 has no indegree
- shortest path matrix SP is $n \times n \times (n \times 1)$
- time complexity is $O(n^3)$

L5.5 Min. Cost Spanning Trees

- roads are damaged, roads restoration acc. to the priority on fiber optic cables example
- retain minimal set of that graph remains connected. Minimal connected graph is a tree. Want a tree that connects all vertices — Spanning-tree



- Spanning trees with cost



→ min. cost spanning-trees

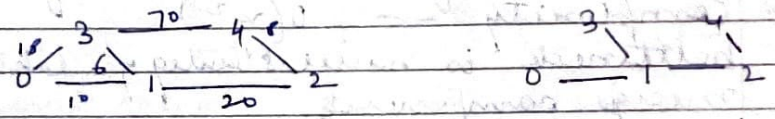
- A tree is connected acyclic graph. ① A tree of n vertices has exactly $(n-1)$ edges. ② Adding an edge to a tree must create a cycle. ③ In a tree, every pair of vertices is connected by a unique path.

[G is connected
G is acyclic
G has $n-1$ edges] Tree

- Prim's algo. — start with smallest & grows
- Kruskal's algo. — scan the edges in asc. order to connect comp. w/o forming cycle

L5.6 Min. cost spanning trees (Prim's Algo.)

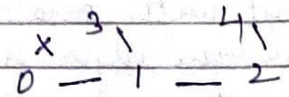
- incrementally growing from 1 edge (which is a tree) → extend the tree



- smallest edge (1,3) → (1,0) → can't add (0,2) → (1,2) → (2,4)
  ↳ this would form a cycle
- correctness — minimum spanning tree lemma
- in the implementation, keep track of —
  - visited(v) — is v in the spanning tree
  - distance(v) — shortest dist. from v to the tree
  - tree Edges — edges in the current spanning tree
- we are looking into list → $O(mn)$ → $O(n^3)$ (not good)
- improved
  - nbr(v) — nearest neighbour of v in tree
- very similar to Dijkstra's algo
- complexity is $O(n^2)$ ⇒ kind of greedy strategy
- bottleneck is identifying unvisited vertex with min. distance.

L5.7 MCST (Kruskal's Algo.)

- start with n comp., process edges in asc.



- start with (1,3) then (2,4) → (0,1), → then try for (0,3) skip → then (1,2)
- use disjoint paths and their later form the tree
- correctness follows the same from Prim's
- complexity — $O(n^2)$
- bottleneck is naive strategy label and merge components
- efficient union-find brings the complexity $O(m \log n)$
- it has bottom up approach
- if edge weights repeat, MCST is not unique

Week-6

L6.1 Union - Find Data Structures

- keep track of components and merge them efficiently
- data struc. to maintain collec^n of disjoint sets —
  - find (u) - return set containing u
  - union (u, v) - merge sets of u, v
- a set S partitioned into components $(C_1, C_2 \ldots C_j)$, each $s \in S$ belongs to exactly one C_j.
- always merge smaller component into the larger one
- individual merge operations can still take time $O(n) \Rightarrow$ more careful accounting
- Union() $\xrightarrow[\text{(amortized)}]{\text{complexity}} O(\log n)$
- Kruskal's Time → $O((m+n) \log n)$

L6.2 Priority Queues

- Job scheduler — maintains a list of pending jobs with their priorities
- when the processor is free, scheduler picks up the job with max. priority and schedules it → also new jobs keep coming up
- unsorted list → insert (easy), delete (hard
  sorted list → vice - versa
- keep the track of size of each row → insert into the 1st row that has space → takes time → $O(\sqrt{n})$ → insert()
- max in each row is the last element → identify the max → delete it → $O(\sqrt{n})$

- maintain a spl. binary tree - heap
  → height $O(\log N)$

L6.3 Heaps

- binary trees — values are stored as nodes in a rooted tree, each node has upto 2 child (L child & R child) → order is impt.
- other than root, each node has a unique parent, leaf node — no children
- size — no. of nodes
- height — no. of levels
- binary tree filled level by level, L → R
- non eg.
  - no holes allowed | struc. problem
  - can't keep a lvl. incomplete | struc. problem
  - heap prob. is violated | int. problem
- complexity → insert() ∝ height of tree
- no. of nodes at lvl. j is 2^j
- if we fill k levels,
  $2^0 + 2^1 + 2^{k-1} = 2^{k-1} - 1$ Nodes
- if we have N nodes, at most $(1 + \log N)$ levels → insert() is $O(\log N)$
- delete max → max. value is at the top of the tree → after 1 value, tree shrinks → restore the heap property downwards
  Complexity → $O(\log N)$
- building a heap — heapify() — $O(N)$

L6.4 Using Heaps in Algos

- heaps can be represented as list / array
- updating values in a min-heap
- updates are $O(\log N)$
- maintain 2 additional dictionaries
- Dijkstra's algo — using min-heap
- heap sort → start with an unordered list → $O(n \log n)$

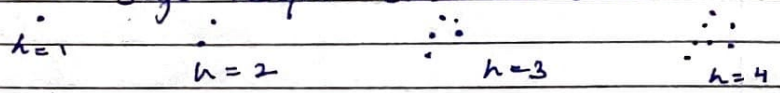
L6.5 Search Trees

- sorting is useful for efficient searching, the data ds dynamically
- insert / delete takes time $O(n)$
- ∴, move to tree struc., like heaps for priority queue
- Binary search Tree — no duplicates, each node has a value and pointers to its children
- worst case complexity — height of the tree
- time — $O(\log N)$

Week-7

L7.1 Balanced search trees

- search trees – find(), insert & delete() all walk on a single path
- worst case – height of the tree
- balanced trees have height $O(\log n)$
- balance – left & right subtrees should be equal. 2 measures – size & height ⇒ this is possible for comp. binary trees
- self.height() – no. of nodes on longest path from root to self
- AVL – Adelson-Velskii, Landis (Height balance trees)
- min. size height-balanced trees



- $S(h)$, size of smallest height-balanced tree of height h
- Fibonacci → $F(0)=0, F(1)=1, F(n)=F(n-1)+F(n-2)$
- $S(h)$ grows exponentially with h
- balanced tree – slope is $\{-1, 0, 1\}$
- t.insert(a), t.delete(a) → slope from $+2$ to -2
- left rotation – converts slope -2 to $\{0, 1, 2\}$
- rebalance each time the tree is modified
- automatically balances bottom up

L7.2 Greedy Algos – Interval Scheduling

- need to make a seq. of choices to achieve a global optimum
- always moves forward
- at each stage, make the next choice based on some local criterion
- drastically reduces space to search for sol^n

Eg.
- Dijkstra's algo., Prim algo.
- IITM has spl. indic classroom for delivering online lectures, slots may overlap, so not all booking honoured, so how to maximize the no. of teachers
- take the 1st slot with any local strategy
- greedy strategy is optimal
- greedy algo – takes local optimum choices to get a global optimum
- it never reconsiders its choice
- just prove that local strategy is globally optimum
- interval scheduling – many natural greedy strategy
- correct strategy needs a proof
- greedy sol^n stays ahead! step-by-step of any optimal sol^n

L7.3 Greedy algos – Minimizing Lateness

Eg. of IITM has single 3D printer
- each user will get access to the printer, but may not finish before deadline
- minimize the maximum lateness
- try to finish first shortest jobs
- or give priority to request with smaller slack time.
- renumber the jobs so that they are sorted by deadline
- There is an optimal schedule with no idle time.
- eliminate idle time by shifting jobs earlier

⇒ this can reduce lateness

- exchange argument & inversions.
- any 2 schedules with no inversions and no idle time have same max. lateness
  → no inversions, idle-time - only diff can be the order of req. with same deadline
  → reordering jobs with same deadline produces same lateness
- there is an optimal schedule with no inversions & no idle time
- schedule req. with start times $t(i)$ and deadlines $d(i)$, to minimize max. lateness
- simple greedy algo. - $O(n \log n)$
  - sorts the req. - $O(n \log n)$
  - reads off schedule - $O(n)$
- correctness by exchange argument.
  - consider an O optimum argument
  - transform it ⟶ greedy soln.

L7.4 Greedy algo — Huffman coding

- efficient communication
- variable length encoding ⟶ morse code
  Eg. e is 0, t is 1, a is 01

x	a	b	c	d	e
E(x)	11	01	001	10	000

- total msg length -

$$\sum_{x \in A} n \cdot f(x) \cdot |E(x)|$$

- avg. no. of bits / letter in encoding

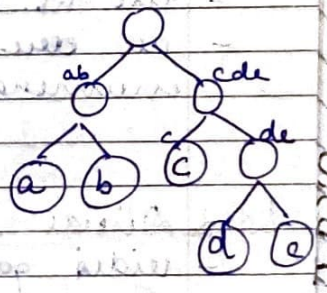
$$\sum_{x \in A} f(x) \cdot |E(x)|$$

- any optimal prefix code produces a full tree
- full tree - each node has 0 or 2 children
- in an optimal tree, for any leaf at max. depth, its sibling is also a leaf
- in an optimal tree, if leaf labelled x is at smaller depth than leaf labelled y. $f(x) \geq f(y)$
- Huffman's algo.

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

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

x	ab	cde
f(x)	0.57	0.43



- Huffman coding is greedy
  (i) recursively combine letters with lowest freq.
  (ii) locally optimal choice
  (iii) never go back & do other pairing
- Claude Shannon - info. theory

Week-8

L8.1 Divide & Conquer Counting inversions -

- = break up a problem into disjoint subproblems
- combine the subs efficiently
- Eg. merge sort, quick sort
- recommending systems = online services recommend items to you, compare your profile with others
- so how to compare profiles?
- no. of inversions ranges from 0 to $n(n-1)/2$ — measure of dissimilarity
- recurrence is similar to merge sort

$$T(0) = T(1) = 1$$
$$T(n) = 2T(n/2) + n$$
$$T(n) = O(n \log n)$$

- the no. of inversions can still be $O(n^2)$
- we are counting them efficiently without enumerating each one of them

L8.2 Divide & conquer closest pair of pts.

- video game, find closest pair of objects
  - n objects ⟶ $O(n^2)$
- points p in 2D ⟶ $p = (x, y)$
  $d(p_1, p_2) = \sqrt{(y_2-y_1)^2 + (x_2-x_1)^2}$
- we have to find the closest pair
- the brute force ⟶ $O(n^2)$
- P_x & P_y sorted on x and y coordinate
- splitting helps to take time $O(n)$
- now, have to combine the recursive solutions ⟶ we can't have 2 pts inside the same box
- linear scan
- overall ⟶ $O(n \log n)$
  - ↳ recurrence (like merge sort)

L8.3 Integer multiplica^n : Divide & conquer

- form partial products ⟶ add them up
- works the same in any base — eg. binary
- to multiply 2 n-bit nos ⟶ $O(n^2)$
  - ⟶ n partial prod.
  - ⟶ add to get cumulative sum
- each partial prod. seems 'necessary'
- Karatsuba's algo. ⟶ fast - multiply algo.
- $T(1) = 1$
- $T(n) = 3T(n/2) + n$ | $a^{\log n} = n^{\log a}$
  - ↳ this forms the geometric series
- $O(n^2)$ ⟶ $O(n^{1.59})$
  - divide & conquer reduces the time complexity of integer multiplica^n
- 1950, Andrei Kolmogorov ⟶ multiplica^n could not be done in sub-quad. time
- Karatsuba ⟶ came with divide & conquer algorithm
- analysis simplifica^n by Donald Knuth.

L8.4 Divide & conquer: Recursion Trees

- complexity $T(n)$ is expressed as recurrence
- Binary Search : $T(n) = T(n/2) + 1$, ⟶ $O(\log n)$
- Merge Sort : $T(n) = 2T(n/2) + n$, ⟶ $O(\log n)$
- recursion-tree — rooted tree with 1 node for each recursive subproblem

- value of each node is spent on the subproblem excluding recursive calls
- on an input size n,
  (i) $f(n)$ is the time spent on non-recursive work
  (ii) a is the no. of recursive calls
  (iii) each recursive call works on a subproblem of size n/c

$$T(n) = aT(n/c) + f(n)$$

- root of recursion tree for $T(n)$ has value $f(n)$
- Each node at lvl. d has value $f(n/c^d)$
- leaves correspond to the base case $T(1)$
- level i has a^i nodes, each with value $f(n/c^i)$
- tree has L levels, $L = \log_c n$
- Total cost is $T(n) = \sum_{i=0}^{L} a^i \cdot f(n/c^i)$

L8.5 Divide & Conquer: Quick Select
- find the k^{th} largest value in a seq. of length n
- sort in desc. order & look at pos^n k → $O(n \log n)$
- if we can find median in $O(n)$, quick sort becomes $O(n \log n)$
- recurrence is similar to quick sort

$T(1) = 1$

$T(n) = \max(T(m), T(n-(m+1))) + n$,

- worst case → $T(n)$ is $O(n^2)$
- if pivot is within a fixed fracn, then quicksort is $O(n \log n)$
- other method → median of medians
- used median of block medians to locate pivot for quickselect
- median of block medians helps to find a good pivot in $O(n)$
- selecn becomes $O(n)$, quicksort becomes $O(n \log n)$
- C.A.R Hoare described quickselect in the same paper as in quicksort
- 1973 — median of medians

L8.6 Implementaⁿ of Quick Select & Fast Select Algo.
- median is not exact but still its better
- Quick select → 9999, 7 sec.
  → 1, 0.005 sec.
- Fast select → 9999, 0.015 sec.

Week 9

L9.1 Dynamic Prog:
- inductive def.
→ Factorial fact(n) = n * fact(n-1)
→ insertion sort
- sol^n to original problem can be derived by combining sol^ns to subproblems
- interval scheduling problem
- each subset of bookings is a subproblem
↳ given n bookings, 2^n subproblems
- greedy strategy looks at only a small frac^n of subproblems
- weighted interval scheduling → seems as before, but each req. comes with a wt.
- inductive sol^n generates same subproblem at diff. stages
- naive recursive implementation recalculates each instance of subproblem from scratch
- Memoization and Dynamic prog.

L9.2 Memoiza^n
- build a table of values already computed
→ memory table
- memoiza^n - check if the value to be computed was already seen before
- store each newly computed value in a table
- look up the table before making a recursive call
- computa^n tree becomes trivial

- Dynamic prog. — anticipate the struc. of subproblems
→ derived from inductive def^n
→ dependencies form a DAG
- solve subproblems in topological order
→ never need to make a recursive call

L9.3 Grid Paths
- manhattan grid
- one-way road, only up & right
from (0,0) to (m,n).
- in gen. (m+n) segments from (0,0) to (m,n)
→ out of 15, exactly 5 are right moves, 10 are up moves
- inclusion-exclusion → counting is messy
- inductive formula^n

L9.4 Common subwords & subseq.
- given 2 strings, find the lengths of the longest common subword
- subproblem dependancy
- subsequence → can drop some letters in between
- analyzing genes
- diff command in Unix/Linux
- again O(mn), using DP

L9.5 Edit distance

- document similarity → insert a charac.
  delete a charac.
  substitute 1 charac by another
- minimum no. of edit operⁿ needed
  Eg. 24 charac. inserted, 18 deleted & 2 subs.
  so edit dist. at most 44.
- also c/d Levenshtein distance
- applicaⁿ → suggestions for spelling correc", genetic similarity of species
- edit dist. & LCS (longest common subsequence)
- LCS equivalent to edit distance w/o substitutions
- Edit distance - transforms $w \mapsto v$
- again $O(mn)$ using dynamic prog. bei memoiza".

L9.6 Matrix multiplicaⁿ

- multiply matrices A, B

$$AB[i,j] = \sum_{k=0}^{n-1} A[i,k]\, B[k,j]$$

- dimensions must be compatible
  A: m×n, B: n×p respectively
  AB: m×p
- computing each entry in AB is $O(n)$
  Overall $O(mnp)$
- matrix multiplicaⁿ is associative
  $ABC = (AB)C = A(BC)$
  ↳ breaking doesn't change ans. but the complexity does changes due to it.
  Eg. A: 1×100, B: 100×1, C: 1×100
  See the diff btw A(BC) & (AB)C
  20000 steps / 200 steps
- in DP, diagonal entries are base case, fill matrix by diagonal, from main diagonal
- fill the table → $O(n^2)$ + filling each entry → $O(n)$
  Overall → $O(n^3)$

Week-10

L10.1 String Matching

- searching for a pattern is a fundamental problem when dealing with text
  - → editing a document
  - → ans. an internet search query
  - → looking for a match in a gene seq.
- find every pos^n i in t such that $t[i : i+m] = p$
- nested loop → brute force
- nested search bails out at first mismatch
- worst case is $O(nm)$
- don't need to check all of t to search for all occurrences of p.
- Formalized in Boyer–Moore Algo.

L10.2 Boyer–Moore Algorithm

- Boyer, Moore, 1977
- t = "which finally halts. at that point"
  p = " at that
- skip over – speed up
- single scan, rightmost value is recorded
- nested loop, compare each segment $t[i : i + len(p)]$ with p
- worst case still remains $O(nm)$
- w/o dictionary, computing last is a bottleneck, complexity is $O(|\Sigma|)$
- this algo.
  - sublinear
  - avg. no. of comparisons is 0.24 / char.
  - performance improves as pattern length ↑
  - in practice, Grep in Unix.

L10.3 Rabin–Karp algo

- reducing string matching to arithmetic
- $\Sigma = \{0, 1, \ldots, 9\}$
- any string over Σ can be thought of as a no. in base 10
- pattern p is an m-digits
- converting a string to a no.
  - → can convert a block $t[i : i+m]$ to an integer x_i in one scan
  - computing x_i ... → $O(mn)$
  - for $\Sigma = \{a_1, a_2 \ldots a_k\}$ treat each letter as a digit in base k
  - in prac:, for realistic k, the numbers are too large to work with directly
  - instead do all computation and comparisons modulo a suitable prime q
  - false positives
  - preprocessing time is $O(m)$.
  - worst case for general alphabets is $O(nm)$
  - in prac. no. of spurious matches will be small
  - thus, overall time $O(n+m)$ or $O(n)$

L10.4 String Matching using automata

- traditional string matching → $O(nm)$
- can use intelligently. re-use partial matches
- to use it, use computed values

- precomputing longest matches as a graph
- pattern p of length m, graph with $(m+1)$ nodes, edges describe how to extend the match
- graph → finite state automation
  - → nodes are states
  - → edges are transitions
- build an automaton to keep track of longest matching prefix
- using this, we can do it in $O(n)$
- bottleneck is precomputing the automaton
- do this in time $O(m)$ → Knuth, Morris, Pratt

L10.5 Knuth, Morris and Pratt Algo

- to compute the automaton for p efficiently
- match p against itself
- analysis of fail computation
- this algo efficiently computes the automaton describing prefix matches in the pattern p
- complexity of preprocessing fail function → $O(m)$
- can check matches in $O(m+n)$
- KMP → $O(m+n)$
- the Boyer-Moore algo is faster in practice, skipping many functions.

L10.6 String Matching: Tries

- it often involves search a large fixed body of text
- collected works of Shakespeare
- comprehensive set of ref. manuals
- genetic data
- make multiple queries on this text
- preprocess this text to make the search easier
- A trie is a spl. kind of tree → from 'information retrieval'
- rooted tree → children of node have distinct labels
- each maximal path is a word
  - → one word shouldn't be a prefix of another
  - → add special symbol at the end of word
- build a trie T from a set of words S with s words and n total symbols
- basic prop. →
  - → height of T is max $\text{len}(w)$ $w \in S$
  - → a node has at most $|\Sigma|$ children
  - → the no. of leaves in T is s
  - → the no. of nodes in T is $n+1$, plus s nodes labelled $
- can maintain auxiliary info. for each word
- suffix trie → expand S to include all suffixes
- more ques can be ans. using suffix trie
  - → a substring ?
  - → a suffix of s ?
  - → no. of times w occurs as a substring of s
  - → longest repeated substring of s

- how big can a suffix trie be for s of length n?
  → no. of nodes prop. to n? → Yes, a^n
  → " " " " " n^2? Yes, $a^n b^n$
- main drawback of a trie is size

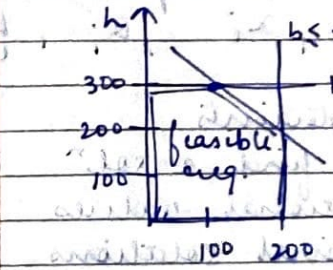
L10.7 Regular Expressions

- what if we want to look for a pattern?
  → misspelling
  → check for a subsequence
- anchoring patterns
  - our convention is now that p matches anywhere in the string
  - to match the start ^p
  - to match the end p$
- KMP algo built a finite state automaton for the longest prefix matched by a pattern
- the automaton we built had a linear structure, with a single path from start to finish
- the set of words that automata can accept are called regular sets.
- the sets we can describe using patterns are exactly the same as those that can be accepted by automata
- our patterns are cld regular expressions
- python provides a library for matching regular expressions

Week-11

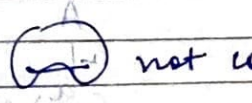
L11.1 Linear Prog.

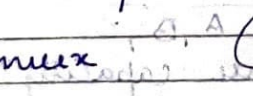
- computational tasks involve optimizn
  - shortest path
  - min. cost s.t.
  - longest common subseq.
- LP — constraints & objectives to be optimized are linear functions
- Eg. maximize profits
  b → boxes of apples, h → boxes of halwa
  Profit : 100b + 600h
  max. → 100b + 600h
  $b \leq 200$
  $h \leq 300$
  $b + h \leq 400$
  $b \geq 0$, $h \geq 0$



  h = 300
  b = 100
  the ans. lies on the corner (any)
  C = 190000

- simplex algo. — can be expo., but efficient in prac.


not convex


convex

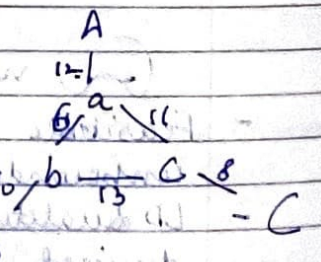
- Feasible reg. is convex, may be empty or unbounded
- LD Duality — This is always possible. We devised an upper bound on the obj. thru a linear combinatn of constraints

L11.2 LP Produc^n Planning

- can always construct a linear prog.
- Produc^n planning
  - 30 employee each 20 carpets / month
  - salary ₹ 20,000 labour → ₹ 1000 / carpet
  - $d_1 \ldots d_{12}$ demand from Jan to Dec.
  - overtime → 80% extra, limit is 30% / worker
  - hiring 3200 / worker firing 4000 / worker
  - surplus storage cost → ₹ 80 / carpet
  - workers → w_i, $w_0 = 30$; carpets → x_i; overtime → o_i; hired/fired → h_i, f_i
  - formulate the constraints
  - Run the simplex & find a sol^n
  - optimum has fractional values
  - handle the fractional solutions
  - ⇒ Integer LP → computationally intractable

L11.3 LP Bandwidth Allocation

- 3 users, A, B, C
- link have capacity constraints
- each connec^n should have at least 2 Mbps.
- direct & indirect connections are allowed
- allocate bandwidth to maximize the revenue



- modelling strategy doesn't scale well
- need to have a better approach to analyze such network flows

L11.4 Network Flows

- network of pipelines
- ship asap from s → t
- Network : graph $(G) = (V, E)$
- special nodes : s (source), t (sink)
- each edge e has capacity c_e
- flow : f_e for each edge e
  - $f_e \leq c_e$
  - at each node, sum of incoming flows = sum of outcoming flows
- LP formula^n
  var. f_e for each edge e
- simplex explores vertices of feasible reg. to solve LP
- Ford - Fulkerson algo.
  start with 0 flow
- residual graph - for each edge e with capacity
- certificate of optimality - max flow - min cut theorem
- in this algo, choose augmenting paths wisely, use BFS to find augmenting path

L11.5 Reductions

- each instruction is willing to teach a set of courses

- bipartite matching
  - $V \rightarrow V_0, V_1$
  - all edges from V_0 to V_1
  - matching: subset of edges so that no 2 of them share an endpoint
- reduce^no
  - we want to solve A
  - we know how to " B
  - convert input A to B
  - A reduces to B
  - can transfer efficient sol^n from B → A
  - but pre & post processing must be efficient
  - both should be in polynomial time
- bipartite reduces to max flow
- max flow reduces to LP
- no. of vars., constraints is linear in the size of graphs
- reverse interpretation is also useful
- big hammers
  - lp & network flows are powerful tools

L10.6 Intractability: Checking Algo.

- good algo — have polynomial time algo.
- search space for sol^n is expo.
- brute force — scan expo. possibilities & choose the best
- large class of 'natural' problems where shortcut isn't available
- generating vs checking
- checking algo. C for problem P
- boolean satisfiability
  - x_i (neg^n) $x_i \vee x_j$ (or) $x_i \wedge x_j$ (and)
- conjunc^n of clauses $(C_1 \wedge C_2 \wedge \dots \wedge C_k)$
- each clause forces a unique value
- Eg. Traveling Salesman
  - a network of cities with distances btw each pair, find the shortest tour that visits each city exactly once & also every city has to be visited
- V is an independent set of size k iff V/V is a vertex cover of size N−k
- indep. set & vertex cover reduce to each other

L11.1 Intractability of P & NP

- NP — factoring^n, satisfiability, traveling salesman, vertex cover, indep. set
  - C is able to check
  - → Non-deterministic Polynomial Time
  - → guess a sol^n & check it
- P → neg. polynomial time algo. (worst case)
  - C is included in NP — generate a sol^n & check it
- P ≠ NP (or) NP
- problems are inter-reducible
- Cook-Levin Th. → Every problem in NP can be reduced to SAT
- SAT is said to complete for NP

Week-12

12.1 Summary Week 1 → 3

- timing our code → time library
  - time.perf_counter()
- analysing an algo → time & memory
  - T(n) space complexity
  - func^n of input size (n)
- upper bound on worst case gives us an overall guaranteed perf
- f(n) is O(g(n)) means g(n) is an upper b
  " is Ω " " " " is lower bo
  " is Θ " " " both (optimal sol^n)
- orders of magnitude
- calculating complexity → type → iterative
  → recursive
- searching algo.
  - Linear
  - Naive
  - check every element 1 by 1
  - worst when element not in list
  - Best O(1)
    Avg. O(n)
    Worst O(n)
  - Binary
  - sorted list
  - compare with mp. element
  - halve the list till list becomes empty
  - $T(n) = T(n/2) + 1$
    O(1)
    O(log n)
    O(log n)
- selec^n sort
  - intuitive algo
  - swapping elements to avoid 2nd list
  - $T(n) = n(n+1)/2$
  - time complexity remains same
- insert^n sort
  - insert into the list $T(n) = n(n+1)/2$
  - time complexity varies based on req element
- Merge sort
  - divides the list into 2 halves and sort the sublist & then combine
  - $T(n) = 2T(n/2) + n$
- Quicksort
  - choose a pivot (1st element), partition into upper & lower, pivot goes in between, sort the partitions recursively. This places an in-sort

	Selec^n	Insert^n	Merg	Quick
Best	$O(n^2)$	$O(n)$	$O(n \log n)$	$O(n \log n)$
Avg.	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Worst	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n^2)$
In-Place	Y	Y	N	Y
Stable	N	Y	Y	N

- List
  - flexible length
  - scattered memory
  - need to follow links
  - insert^n & del is easy
  - swapping takes 3 steps
  - list in python → maps to array
    l.append & l.pop → O(1)
    insert^n & del^n → O(n)
- Arrays
  - fixed
  - contiguous block
  - random
  - expensive
  - linear time
- dict implemented as hash table (hash func) array
  need a strategy to deal with collision
  - closed – probe for free space in array
  - open – each slot pts. to a list of key:value pairs

L12.2 Summary wk 4-6

- graph → relationship bw entities
  - directed or undirected
  - paths are seq. of connected edges
- reachable → from u to v
- $G = (V, E)$; $|V| = n$, $|E| = m$
  if G is connected, m varies $(n-1)$ to $n(n-1)/2$
- Adj. matrix → n×n matrix
  " list → dictionary of list
- BFS – explores the graph, level by level, queue, adj. matrix $(O(n^2))$, list $(O(m+n))$
- DFS – uses a stack (suspend) same complexity as BFS
- paths by BFS is shortest paths in terms of no.
- find strongly connected points
- DAGs → dependent dependencies
  ① Topological Sorting
  ② longest paths – $O(m+n)$
- Single source shortest (Dijkstra's algo)
  - greedy strategy
  - complexity $O(n^2)$
  - not work with –ve weight
  - min heap → $O((m+n)\log n)$
- single source + (–ve) wt. (Bellman Ford)
  adj. matrix – $O(n^3)$
  adj. list – $O(mn)$
- all pair shortest paths (Floyd–Warsh)
  complexity – $O(n^3)$
- Trees – connected acyclic path
- Spanning tree – MCST (multiple), but cost will be unique → Prim's
  → Kruskal

- Prim's – implementation similar to Dijkstra
  Cs complexity – $O(n^2)$
- Kruskal's – $O(n^2)$ due to naive handling of components, $O(m \log n)$ using union-find data struc.
- Union-find
  - Union() – log n
  - Find() – O(1)
- Priority Queues
- Heaps – binary tree, no holes
  insert / delete – $O(\log n)$
  heapify – $O(n)$
- Search trees
  Cs worst case – an unbalanced tree with n nodes may have height $O(n)$
  find, insert, delete – all might down to same path
- Balanced search trees – AVL trees
  find, insert, delete – $O(\log n)$

L12.3 Summary wk 7 → 9

- greedy algo – global optimum, never go back, needs a proof of optimality
  Eg. Dijkstra, Prim, Kruskal's algo
- Applications of greedy algo.
  (i) Interval scheduling – IITM classes
  (ii) minimizing lateness – 3D printer
  (iii) Huffman coding – $O(k \log k)$

- Divide & Conquer — eg. merge, quick sort
  Eg: counting inversions — nonic product
  $T(n) = 2T(n/2) + n = O(n \log n)$
- closest pair of pts — video game
  $O(n^2) \rightarrow O(n \log n)$
- integer multiplication
  Naive — $O(n^2)$
  Karatsuba's — $O(n^{1.59})$ or $O(b^{\log 3})$ $\left[3T(n/2)+n\right]$
- quick select — $O(n \log n)$
  ↳ median of medians → fast-select
- recursion trees
  $T(n) = aT(n/c) + f(n)$
- DP → sol^n of problem can be derived from sol^n of subproblems
  ↳ solve subproblems in topological order
- Memoization → build a table of values already computed — memory-table
- Grid paths — up & right
  $O(mn)$ — DP
  $O(m+n)$ — memoization
- common subword & subseq. & edit dist.
  complexity — $O(mn)$
- matrix multiplication — $O(n^3)$

L 12.4 Summary week 10-11

- String matching — searching for a pattern while dealing with text
  → $O(mn)$

- Boyer-Moore algo → works well in prac. but worst case is $O(nm)$
- Rabin-Karp algo → $O(n+m)$ or $O(n)$
- Automata → $O(m^3 \cdot |\Sigma|)$
- Knuth-Morris-Pratt Algo → easily calculates this automata → $O(m+n)$
- Tries (info. retrieval) → rooted tree, each maximal path is a word ⟹ suffix trie
- Regular expressions