* Big-Oh
O(.):

T(n) is said to be O(f(n)) if there exist constants c, n_0 such that

$\forall n \geq n_0 \quad T(n) \leq c \cdot f(n)$.

Asymptotic upper bound.

Omega
$\Omega(.)$:

asymtafic lower bound.

Theta
$\Theta(.)$:

$\exists\ n_0, c_0, c_0'$

Such that $\forall n \geq n_0$; $c_0 f(n) \leq T(n) \leq c_0' f(n)$

order is same.... ☺.

---

1) Interval Scheduling problem:-

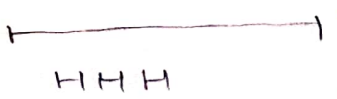
aset $\{I_i = (s(i), f(i))\}$ of jobs.

Only one computer.

Schedule as many jobs as possible.

Greedy-1:-

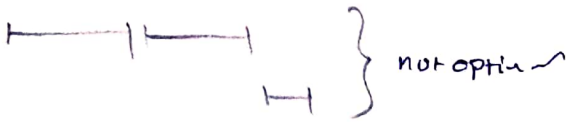
choose earliest starting time.



not optimally solved. Wrong

Greedy-2:-

choose shortest jobs.



not optim~

Greedy-3:-

Select earliest finish time.

(keep the machine empty as soon as possible)

this works!

Why?

let A denote the set of jobs by Algo.

O denote the optimal set of jobs.
i.e. the largest set possible.

* to complete our proof; we gotta show $|A| = |O|$

ie; $|O| \geq |A|$ since O is optimal ✓.

Proof: let $i_1, i_2 \ldots i_k$ be the jobs of A.

$j_1, j_2 \ldots j_m$ be the jobs of O.

1) Assume an optimal solution.
2) Show that our Algo is not worse than the optimal soln.
its enough.

Lemma:
$\forall t \leq K$; $\text{finish}(i_t) \leq \text{finish}(j_t)$

Proving Lemma:-

$t=1$ ✓ (By def^n of Algo.)

Say; true till $r-1$.

i.e. $f(i_{r-1}) \leq f(j_{r-1})$

& we know $s(j_r) \geq f(j_{r-1})$

$\therefore s(j_r) \geq f(i_{r-1})$
start

$\therefore$ when Algo chooses r^{th} term;

j_r is available.

but i_r gets chosen

$\Rightarrow f(i_r) \leq f(j_r)$

$\therefore$ if a job is picked by O; it is also pickable by A.

Runtime:- $O(n \log(n))$ for sorting.

$O(n)$ for a scanning & picking.

$\therefore O(n) + O(n \log n) \equiv O(n \log n)$ ✓

→ Minimum Spanning Tree:- (Hence, same for maximum spanning tree).

* A connected graph $G(V,E)$ is given with a cost function $C(E) \rightarrow Z^+$;

(Assume distinct costs).

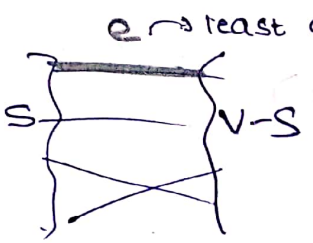
we need to find subgraph T, such that

1) T is least cost.
2) T covers all vertices } defⁿ for MST.
3) T is connected.

⇒ MST: minimum spanning tree for G.

→ The Cut property:-

In a connected $G(V,E)$ divide vertices into S, V/S.



let 'e' be the least cost edge b/w S, V-S. then

(e) should be in MST. | Proof by contradiction.

Algos:

1) Kruskals Algo:- $O(n \log(n))$

1. Sort the edges in increasing cost.
2. take first edge.
3. if $T \cup \{\text{new edge}\}$ doesn't have a cycle; $T \leftarrow T \cup \{\text{new edge}\}$
4. do this till all edges complete.

(greedy)

Proof that this will give me MST:-

→ no cycles finally ✓

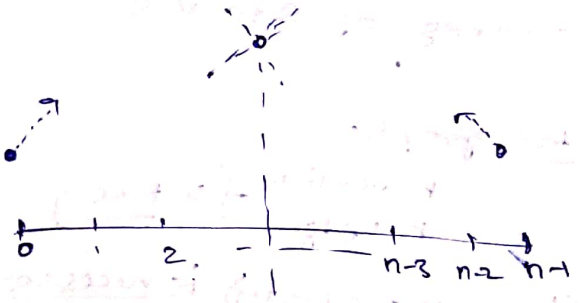
→ connected finally ✓ (∵ we add all edges which don't give a cycle).

→ The added edge will be a min. cost edge b/w S & V-S where S is the set of points connected to V; in the

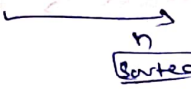
∴ edge is necessary in MST.

Tut: 1

3)



4)



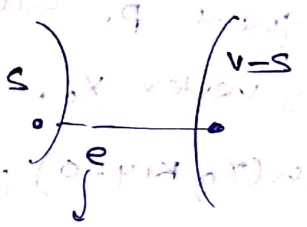
a number per

$\lceil n/2 \rceil$ times.

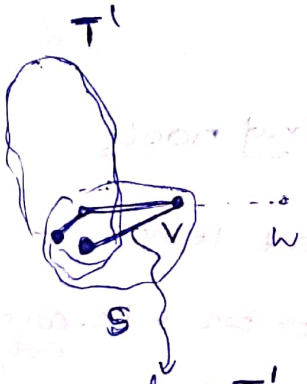
Tut 2:

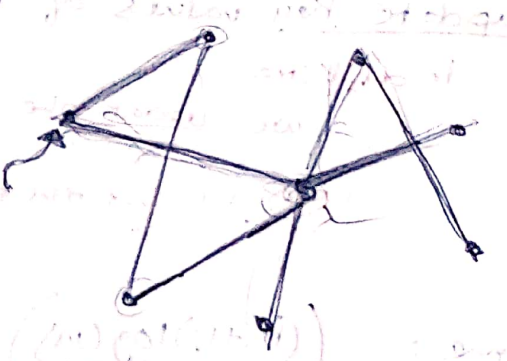
Kruskals Algo correctness:-

cut property



least cost edge.
gotta be included in MST!





Prims Algo:-

n.logn

n.logn.

mlogn log

+n.logn

least

block

least.?

priority queue.

2) Prim's Algorithm:-

1. choose a vertex; & its min-edge.
2. Now; 2 vertices is S. choose min edge of S to V-S.
   :
3. Finally; we get a connected graph;
   n vertices, n-1 edges } → tree !

   & every edge; (by cut property) is necessary.

∴ MST ✓✓

Time:. & implementation:-

1. Priority_queue P. outputs the node with the least key.
2. arbitrary vertex x_0.
3. P.push(x_0, key=0) ; P.push(V-{x_0}, key=∞)
4. while node is remaining;

   cur_node = P.pop( ) (least key node is popped)

   update key values of neighbouring nodes

   in dijkstra;
   we update the key to reflect length of shortest path.

   now; we set the key; to remember min_cost edge connecting node.

∴ time = $((m+n)\log(n))$

edges. vertics. for operations on priority queue!

NOTE:

priority queue has push, pop, change_key all in log(n).

underneath is binary heap or BST.

Tut 2 :-

Huffmann Coding:-

> given a file with text in english.

> convert into binary; using as few bits as possible; uniquely decodable.

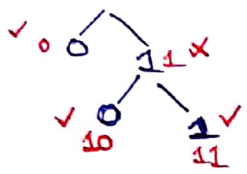
use prefix free!

letters	a	b	c	d	e
frequency	100	3	5	45	20

Idea-1:-
5 → 3 binary bits.
alphabet.
use
000 001 010 011 100 ....
but
maybe, we could use less than 3 bits.. to save cost.

* Use prefix-free encoding:-
(for unique parsing).

0
1



nodes are subsets of alphabets
leaves are alphabets

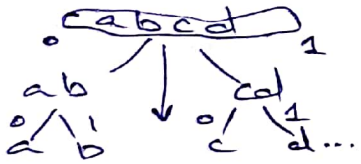
& 1st Greedy:-

Fe?) order mei ascending.

& assin largest f(i)'s element the shortest string.

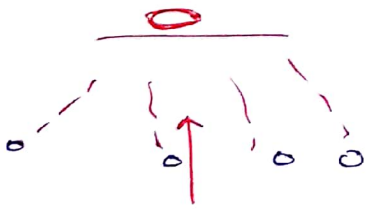
will not work. slides hav counter example.

2nd Greedy:- Top to bottom

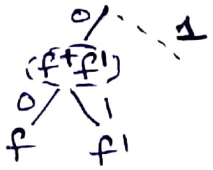


will not work
slides hav counter!

3rd Greedy:- bottom-UPS



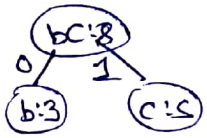
1. Start with two least frequently occuring elements f, f'



give suffixes 0, 1.
(we build from bottom na).

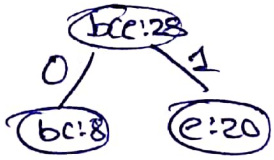
2. Make a new word with freq f+f' & do the same again

Eg: a b c d e
100 3 5 45 20



OPTIMAL

a bc d e
100 8 45 20



Pseudocode:-

Input:- $\Sigma = \{a_1, a_2, \ldots a_n\}$

& f_i for each a_i.

Output:- Binary encoding of letters.

```
1. for i∈[n] do
2.     create a leaf node ai.
3.     Insert into Min heap H; with fi as key.
4. end for
5. while H has >1 element do.
6.     extract the two nodes with lowest key values. u,v
       let fu, fv be keys
       create new node w with fw = fu+fv
       Let w's left child be u; right child be v
       Insert w into H.
   end while
   return H.
```

W is ... a tree in itself.

Correctness:-

1st step of greedy is optimal.

Lemma:-

There exists an optimal tree T^* in which u,v are leaves... in the lowest level.

"exchange argument".

Optimal Substructure property:-

Assuming the first step is optimal; next step is also optimal:-

Lemma:-

T is optimal encoding of Σ. x,y are leaves & z is their parent.

$f_z = f_x + f_y$

$T' = T / \{x,y\}$.

$\Sigma' = \Sigma / \{f_x, f_y\} \cup \{z\}$

Then T' is optimal for Σ'.

Divide and Conquer:-

Paradigm:-

- Solve a task of size n.
- divide into K-subtasks of size n/K.
- recursion on subtasks.
- Combine the answers on subtasks.

$$T(n) = K \cdot T(n/K) + [\text{combining time}].$$

Eg Mergesort

$$T(n) = 2 \cdot T(n/2) + n^1.$$

By master rule:-

$$a \cdot T(n/b) + n^c.$$

$a = b^c$. $\rightarrow (n^c \log n)$ time...

if $a < b^c$; $= n^c$
if $a > b^c$ $= n^{\log_b a}$

→ Integer multiplication:-

input:- 2; n-digit non-negative integers x, y.

compute: $x \times y$.

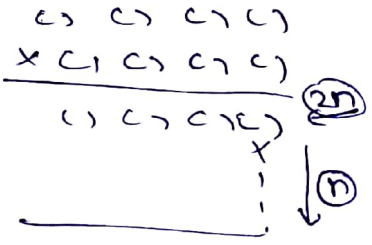
our simple algorithm of high-school.

primitive operations:-

Add two, 1-digit numbers — $O(1)$

Multiply two, 1-digit nos — $O(1)$

insert a zero at end — $O(1)$



$2n^2$ $O(n^2)$.

I wanna do better!

→ Karastuba's algo:-

a → 56 78 ← b
× 12 34 ← d
c

1. compute $a \cdot c$
2. compute $b \cdot d$
3. compute $(a+b) \cdot (c+d)$
4. $w = (a+b) \cdot (c+d) - ac - bd = ad + bc$.
   but no multiplying! just add!
5. our answer:-

$$10^4 (ac) + 10^2 (w) + bd$$

✓

$\therefore$ Mult(u, v)

$$= 10^n \cdot \text{Mult}(a,c) + 10^{n/2} \left(\text{mult}(a,d) + \text{Mult}(b,c) \right) + \text{Mult}(b,d)$$

But!

$$T(n) = 4 \cdot T(n/2) + O(n)$$

$a = 4, b = 2, c = 1$

$\therefore T(n) = n^{\log_b a}$

$= n^2$ Darnit!

Master Theorem:-

let $T(n) = a \cdot T(n/b) + \Theta(n^c)$

$a, b, c \in \mathbb{N}$; $a, b > 1$, $c \geq 0$.

- $T(n) = \Theta(n^c)$ if $a < b^c$
- $T(n) = \Theta(n^c \log n)$ if $a = b^c$
- $T(n) = \Theta(n^{\log_b a})$ if $a > b^c$.

* if I do mult(u, v) $= 10^n \text{mult}(a,c) + 10^{n/2} \left(M(a+b, c+d) - M(a,c) - M(b,d) \right) + M(b,d)$.

$$T(n) = 3 \cdot T(n/2) + O(n)$$

$$T(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1.584})$$

✓ sweet.

How to do 3products, 3 terms?.

a b c → $(10^2a+10b+c)$

d e f. $(10^2d+10e+f)$

$10^4ad + 10^3ae + 10^2af$

$+10^3bd + 10^2eb + 10bf$

$+ 10^2dc + 10ec + fc$

10^4ad

$+ 10^3(ae+bd) + 10^2(af+be+cd)$

$+ 10(ec+bf) + fc$

ad cf

be

$(a+b)(d+e)$ &: $6 \cdot T(n/3)$

$(a+f)(c+d)$ $3\ n/2$.

$n^{\log_3 6}$ $n^{\log_3 6}$.

→ Geometric Algorithm: closest points in plane:-

Trivial: nC_2 values $O(n^2)$.

min.. $O(n^2)$

$\therefore O(n^2)$.

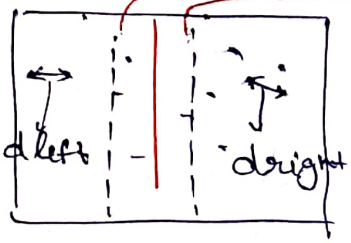
I want better!

1-D:- $O(n\log n)$ → by sorting $O(n\log n)$

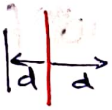
& seeing adjacent points $(O(n))$.

2-D:- Smthing similar, maybe? Nah!. ~~will go n^2; for seeing points.~~

*



divide into 2;
find for them;
& report min of both.
NAH!
Still $\Omega(n^2)$ comparisions are pending!

1. compute dleft, dright.
2. $d = \min(\downarrow, \downarrow)$!
3. in the area:  ; find out if points are closer than d.

Lemma:- let S_y be the points, in the distance 'd' region, sorted in decreasing order of their y-coordinates.

O(n) ✓ (Bcoz we already sort; at starting).

Say $S_y = \langle q_1, q_2, \ldots q_m \rangle$. If the distance b/w some q_i, q_j is $< d$; then $j - i \leq 15$.

meaning; i want $|q_j - q_i| < d$ but $|q_k - q_i| > d$, $k \in \{i+1, \ldots j-1\}$

then; max 9 can manage is $j - i \leq 15$.

∴ now;

O(n) time to move forward from d_{left}, d_{right}.

Algo:-

Given: n points; $P_i(x_i, y_i)$ ....

Output: i, j such that the dist. b/w P_i, P_j is minimum...

P_x → array of points, stored in increasing order of x.

P_y → ———— increasing order of y.

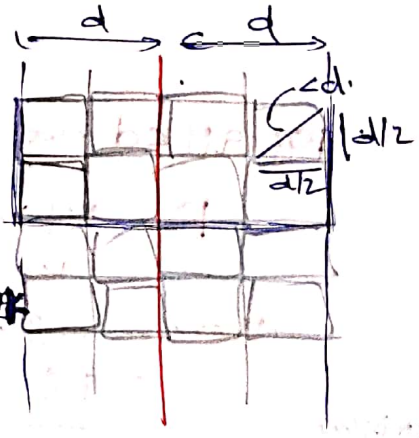
```
closestPair (Px, Py)
    if |Px| = 2 then
        output the dist. b/w points
    end if
    dleft ← closestPair( Firsthalf(Px, Py))
    dright ← closestPair (Secondhalf(Px, Py))
    d = min(dleft, dright).
    let Sy be points from Py; within distance d;
    for i = 1 to |Sy| do
        for j = 1 to 15 do
            d ← min(d, dist. b/w Sy[i], Sy[i+j]);
        end for
    end for
    Output d.
```

Time analysis:- Sorting initially O(nlogn).
distance within d range – O(n).

$T(n) = 2 \cdot T(n/2) + O(n)$ → master's thm: $O(n^{c-1} \log n)$.

proving lemma:

Say that points are farther than d.



∴ points lie uniquely in some box.

could give ≤7 too, I think

if >15;
then it means; atleast 3 rows diff.

$$\Delta y > \frac{3d}{2}$$
$$> d$$

∴ If two points are less than d;
then; for sure;
there exist two
≤15 distant Sy nodes.

if within 15, we didnt find <d;
we wont find it any further.

Think! (later)

But ..... say we found d/100. if we still keep 16;
d/10,000 might be missed....

→ univariate polynomial multiplication:- FFT.

input: $A(x) = a_0 + a_1x + \cdots a_{n-1}x^{n-1}$, $B(x) = b_0 + b_1x + \cdots b_{n-1}x^{n-1}$.

output: $C(x) = A(x) \cdot B(x)$ as coefficient vector.

$$c_k = \sum_{i,j<n,\ i+j=k} a_i b_j.$$ eazy $O(n^2)$ Algo.

for 2 vectors $[a_0, a_1, \cdots a_{n-1}]$ $[b_0, b_1, \cdots b_{n-1}]$;

$[c_0$ to $c_{2n-1}]$ as above is called convolution.

occurs naturally in polynomial multiplication.

can we get $O(n \log n)$?- phew! yea....

lets consider evaluations of $C(x)$ at $2n$ points by multiplying evaluations of $A(x)$, $B(x)$ at these $2n$-points...

& recover $[c_0, c_1, \cdots c_{2n-2}]$ from those evaluations.

But random $2n$ points; evaluation for $A(x) \longrightarrow O(n^2)$ dammit!

- use DIVIDE AND CONQUER for polynomial evaluation

$$A_0(x) = a_0 + a_2x + a_4x^2 + \cdots a_{n-2}x^{n-2/2}$$
$$A_1(x) = a_1 + a_3x + \cdots a_{n-1}x^{n-2/2}$$

$$A(x) = A_0(x^2) + xA_1(x^2)$$

& evaluate $A(x)$ at $2n^{th}$ roots of unity.... (x^{2n-1} (now).

$$w_{j,2n} = e^{\frac{2\pi j}{2n} \cdot i}$$

$$A(w_{j,2n}) = A_0(w_{j,n}) + w_{j,2n}A_1(w_{j,n})$$

assume we have recursively. assume we have it recursively.

$T(2n) = 2T(n) + O(n)$ for all n^{th} roots.

$= O(n \log n)$ (master rule).

∴ evaluation at all roots of unity $\longrightarrow O(n \log n)$

(ii) $C(x) = A(x) \cdot B(x)$

$O(n)$ time.

(iii) interpolation of coefficients....

will it be possible in $O(n \log n)$.

normally, determinant solving, we have $O(n^3)$.

WE! HAVE! used roots of unity!

use that!

not random evaluations!

This will reduce interpolation of C to evaluation of another polynomial $D(x)$.

$$C(x) = \sum_{s=0}^{2n-1} c_s \cdot x^s.$$

Say $D(x) = \sum_{s=0}^{2n-1} d_s \cdot x^s$ & $d_s = C(\omega_{s,2n})$

already evaluated in (ii)

then

$$c_s = \frac{1}{2n} D(\omega_{2n-s,2n})$$

polynomial evaluation! $O(n \log n)$

lol...

FAST FOURIER TRANSFORM

Dynamic Programming:-

Dont do unnecessary work! Save repeating results!
memoization.

1. Fibbonacci:-

```
Fib(K):-
if K=0, return 0;
if K=1, return 1;
return Fib(K-1)+Fib(K-2)
```

Running Time : $T(n) = T(n-1)+T(n-2)+1$

$O(2^n)$ oh lordo!

Lord says :- u dumbwit! Stop those unnecessary calculations. Just remember those results!.

"Those who don't remember the history, are condemned to repeat it".

Memoization:-

Keep values calculated by recurrence; for future usage.

"memoize" them. How about precalcwate all this stuff... to answer instantaneously?

```
Fib(K):-
if Table contains f(K) then
|   return f(K)
end if
if K=0 | 1 then
|   return K;
end if
else
   val ← F(K-1)+F(K-2)
   store val as F(K)
   return val.
```

O(n) Algo!

bcoz; each F(i) is evaluated atmost once.

2. Weighted Interval Scheduling:-

Input:- Foreach $i \in [n]$; start and finish times $(s(i), f(i))$ & weight $w(i) \in N$ (positive weights!)

Output:- Max. weight non-conflicting Jobs.

We've tried! No good greedy strategy!

Wat about recursive kinda algo; which kinda checks all possible combos...

Strategy:-

> sort intervals; based on finish times.

$f(1)\ f(2)\ f(3) \cdots f(n-1)\ f(n)$.

> let $p(i)$ be j; if job-j is the last job before i; & not conflicting with i.

Gotta pre-process i too!

$(n \log n)$ ↳ n x. $\log n$ (n terms; binary search.)

> Now; (the step; which makes us feel, as if we are checking EVERY possibility).

if n^{th} job $\in$ OPT;
recurse on $\{1,2,\cdots p(n)\}$.

if n^{th} job $\notin$ OPT;
recurse on $\{1,2,\cdots n-1\}$.

> might think; this will be 2^n na?

Na! we will discard conflicts while choosing!

Obvious -_-.

> Also; we will choose optimal of remaining.

> so; not 2^n; but might be $n^2, n^3 \cdots$

> what if; I also memoize!

then.... $O(n)$ possible?

But! first; we gotta sort them! $(n \log n)$ (according to something!)

here; by $f(i)$.

Then DP!.

* TO improve; we need to memoize something!

lets have; $OPT(i)$ denote the max. weight for $\{1,2,3,\cdots i\}$

then; our ans. is $opt(n)$.

- Assume jobs are sorted $O(n \log n)$
- we also have $p(i)$ $O(n \log n)$

```
wtIntSc(i)
    if i = 0 then
    | return i
    else if Table[i] is non-empty
    | return Table(i)
    end if.
    Table(i) ← max{ w(i) + wtIntSc(p(i)), wtIntSc(i-1) }
    return Table(i)
```

↓ Looks bad (like; it will be $O(n^3)/n^2$)

but with memoization; it will be $\leq O(n)$. at most. coz; all are calculated once.

→ Correctness of recursion:-

by ind^n:-

$opt(0) = 0$

$wtIntSc(0) = 0$

Say $wtIntSc(i) = opt(i)$ $\forall i < n$

then

by common sense,

$wtIntSc(n) = opt(n)$ ✓

DONE

- Steps for dynamic programming:-

1) Figure out the type of subproblems - for recursion

2) Define a recursive procedure - connect 'n' to lower numbers.

3) Decide on memoization.

4) Verify that dependencies are acyclic. ! Sweet!

5) Analyse time complexity of recursion.
and pre-processing (--).

- 1) Parenthesization
  2) Longest increasing subsequence
  3) Segmentation
  4) LCS : Longest common subsequence.
  5) Edit distance:-

* Sub problems; usually

Suff [i,i]; pre [i,n]; substring [i,j]. ✓✓

Store all these.

3. Parenthesization problem:-

inputs:- string of matrices $A_1, A_2, \ldots A_n$
along with their dimensions $(C_i \times r_i)$

Output:- Find min. cost to multiply them...

Step-1:- Figuring out the subproblems.

the outer matrices

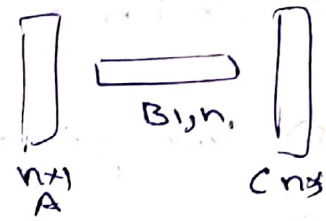
$(A_1 A_2 \ldots A_k) \times (A_{k+1} \ldots A_n)$

( ) ( ) × ( ) ( )

∴ substrings are the sub-problems

use sub[i,j] for $A_i \times \ldots A_j$.

Eg:



n×1 A C n×1

(A×B) ×C
is $O(n^2)$
A×(B×C) is $O(n)$.
wow!

Step-2:- what exactly is the rel^n?

$$Para(i,j) = \min_{i \le k < j} \{ para(i,k) + para(k+1,j) + cost(i,k,j) \}.$$

O(1) computation.
if $A_{m \times n}, A_{n \times h}$,
then cost = ---
O(1) time!

Step-3:- memoization:-

obviously para(i,j)

Starting from small intervals; building larger ones.

$O(n^2)$ → since; every term; atmost once.

Step-4:- Acyclic ✓✓
fine. larger; use smaller.

Step-5:- Time analysis:-

per subproblem, time for "computing" is

time for min, over j=1.

$\simeq O(n)$.

usually, when calcing higher terms, lower terms are available in tables, by then!.

& no. of subproblems $= O(n^2)$

$\therefore$ no. of subproblems x time for each

$= O(n^2) * O(n)$

$= O(n^3)$

4. **Shortest path:-**

description:-

input:- Given a directed graph $G=(V,E)$ and a weight function $W: E \to \mathbb{Z}$ and designated vertices $s, t \in V$.

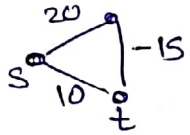
Output:- length of shortest path from s to t.

Note:- weights can be neg. here.

→ DP paradigm:-

1) will dijkstra work here?

NO!



Step-1:- what are my subproblems:-

we need opt(s,t).

So... opt(v,t) is one possibility.

E.. Sure! why not! only if no cycle!

Step-2:- write recursion equn.

otherwise cyclic later! won't hold

$$opt(v,t) = \min_{u \in V;\ (v,u) \in E} \left[w(v,u) + opt(u,t) \right].$$

Step-3:- memoization:-

opt(v,t).

Step-4:- Acyclic nature of recursion:-

I am taking acyclic graph right now.

Step-5:- Time:-

No. of subproblems × time

$= |V| \times |V|$ (at max.)

without cycles

But; think nicely;

$\simeq O(|E|) + O(|V|)$

totalling subproblem's terms.

general check.

could be isolated vertex.

but gotta check na!

PTO-2 pgs

Extra:-

Dijkstra Algorithm:- Give a start node-S; Shortest path to each other node.

Basic graph problems are based on greedy.

Directed graph $G=(V,E)$; start node S.

each edge $e \rightarrow$ length l_e.

S has a path to every other node in G.

Dijkstra's Algorithm (G, l)

Let S be the set of explored nodes ! at any instant;

S is the explored area.

For each $u \in S$, we store distance $d(u)$.

minimum!

Initially $S = \{s\}$ and $d(s) = 0$

while $S \neq V$

priority queues

Select a node $v \notin S$ with atleast one edge from S for which

$$d'(v) = \min_{e=(u,v): u \in S} d(u) + l_e$$ is as small as possible

Add v to S and define $d(v) = d'(v)$

Endwhile

Proof:-

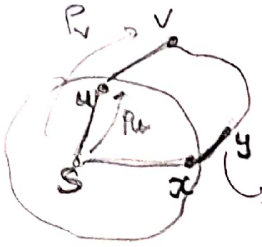
induction on $|S|$.

$|S| = 1$ $d(s) = 0$ ✓

Say, true on $|S| = k$.

then on $k+1^{th}$ iter;

Say we added $(u,v) \in v$ to S.



P_v stays ahead of arbitrary P.

Path P. $\geq P_v$

not added in $(k+1)^{th}$ iter!

we will prove by "stays ahead" argument.

> Holds only for +ve lengths. At any instance; S is explored

$d(u)$ is SHORTEST; for u. even though we didnt explore all vertices!

Priority Queue:-

can efficently

insert element
delete element
change element's key
pop the min key element.

PTO:-

dijkstra algo:-

changeKey in P.queue. — O(log n) : heap based priority queue.

— occurs once per edge. (at most).

∴ =

O(1) × O(log n) for heap....

popMin from P.queue

— occurs 'n' times.

∴ time = n · O(1) + m · O(log n) + n · O(log n)

= O(m log n) = O(E log V) or O((V+E) log V)

?

fibbonaci heap....

Bellman ford again:-

contd:-

Now; with cycles

input:- A directed graph $G=(V,E)$ with weights $w: E \to \mathbb{Z}$ (integers!)

output:- a start node s; terminate node t.

min. weight path from s to t.

Procedure:-

Step-i:- Subproblems?

let $opt(v,i)$ be the minimum weight path from v to t that uses atmost i-edges.

v.v.imp term. not exactly i-edges!.

we want $Opt(s,n-1)$.

u got it, right!

Step-ii:- recursion rel^n:-

$$Opt(v,i) = \min\left\{ Opt(v,i-1),\ \min_{u \in V,(v,u)\in E} \{ Opt(u,i-1) + w(v,u) \} \right\}$$

tricky!

i know!

Step-iii:- memoization:-

opt(v,i)

Step-iv:-

NO cyclic dependencies. (one row down!)

```
ShortestPath(G, s, t)
  M[t,0] = 0
  for v ∈ V\{t} and i ∈ {0,1,... n-1} do
  |   M[v,i] = ∞ = MAX_INT
  end for
  for i = 1 to n-1 do
  |   for v ∈ V (in any order) do
  |   |   Compute M[v,i] using recurrence (*)
  |   end for
  end for
  return M(s, n-1)
```

$O(n^2)$ elem $\times$ $O(n)$ above wala...

$= O(n^3)$

sick!

- A related problem:-

1) Cycle(G, t):-

directed graph. — is there a negative cycle, reachable to t. (= not 'from')

2) Cycle(G):-

is there any negative cycle; reachable to t.

* we call problem-② reducable to problem-①

(put a new vertex to, & draw an edge from every vertex to to; and apply prob-①'s Algo to the new graph).

& to.

wow.

And, how do we solve Cycle(G,t):-

?

Lemma-1:-

There is no negative cycle in G, reachable to t iff

$Opt(v,i) = Opt(v,n-1)$ for each $v \in V$; $\forall i \geq n$.

cuz, if there is a neg. cycle

for some v; $\lim_{i \to \infty} Opt(v,i) = -\infty$.

Lemma-2:-

There is no negative cycle in G, with path to t iff

$Opt(v,n) = Opt(v,n-1)$ for each $v \in V$.

okay ✓ got it!

Buffer.

→ Flow Networks; Max-flow; Min-cut :- [Module-2]

Flow network:-

→ It is a directed graph $G=(V,E)$ with a designated source s and sink t.

→ each edge has a non-negative integer associated.

called capacity (u,v).

flow of (u,v) is what actually flows!

→ there are not inward-edges to source. No outward edges from sink.

* Practical models:- pretty obvious:---

I) 1) $s \equiv$ source of water.

2) junctions are places where no water is stored

3) $t \equiv$ reservoir.

II) current circuit.

III) computer network.

---

* Now, what is FLOW?

→ The flow in network $G=(V,E)$ is a function $f: E \to \mathbb{R}$ that satisfies:-

Capacity constraints:- $\forall e \in E;\ f(e) \leq c(e)$

Flow conservation:- $\forall u \in V \setminus \{s,t\}$,

$$\sum_{v \in V,(v,u) \in E} f(v,u) = \sum_{v \in V,(u,v) \in E} f(u,v)$$

inwards = outwards

except for s,t.

Value of flow $|f|$ = total function values on edges outgoing from s.

Some notations:-

$$f^{\leftarrow}(u) = \sum_{v \in V; (v,u) \in E} f(v,u)$$

$$f^{\rightarrow}(u) = \sum_{v \in V, (u,v) \in E} f(u,v)$$

* hence $|f| = f^{\rightarrow}(s)$

* Flow for a subset of V:- really abused...

for $U \subseteq V$;

$$f^{\leftarrow}(U) = \sum_{u \in U} f^{\leftarrow}(u)$$

$$f^{\rightarrow}(U) = \sum_{u \in U} f^{\rightarrow}(u)$$

} I dont like this definition. Its kind-of lame.

Lemma:-

$$|f| = \sum_{v \in V, (s,v) \in E} f(s,v) = f^{\rightarrow}(s) = f^{\leftarrow}(t) = \sum_{v \in V, (v,t) \in E} f(v,t)$$ fine!

* Max flow problem:-

Given an flow network $G = (V,E)$, designated vertices s,t;

find the maximum flow value acheivable.

An (S,t)-cut:-

- given by $S,T \subseteq V$ such that
  - $s \in S,\ t \in T$
  - $S \cup T \equiv V,\ S \cap T = \phi$.

capacity of a cut:-

given graph $G = (E,V)$ with capacity function $C: E \rightarrow \mathbb{N}$; the capacity of a cut (S,T) is

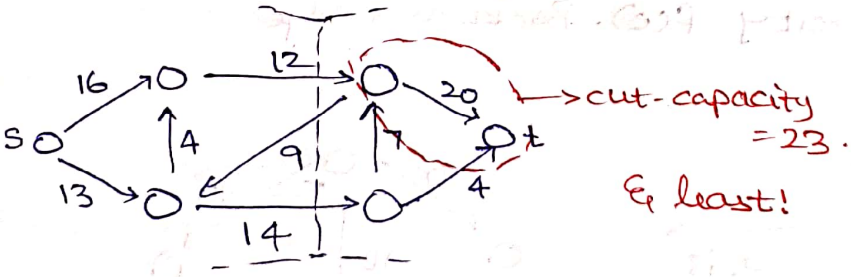
$$cap(S,T) = \sum_{u \in S,\ v \in T,\ (u,v) \in E} c(u,v)$$

only forward edges from S to T.

Min-cut problem :-

Given a flow network $G = (V,E)$ & $C: E \rightarrow \mathbb{N}$; find the (S,T) cut with least cut-capacity.

Eg:



cut capacity $= 12+14 = 26$

→ **weak-duality:-**

let 'f' be any flow in the flow network G, let (S,T) be any (S,t) cut in G. Then $|f| \leq cap(S,T)$. Moreover, $|f| = cap(S,T)$ only if the flow 'f' saturates every forward edge, and avoids every backward edge.

proof:- $|f| = \overrightarrow{f}(S)$

$= \overrightarrow{f}(S) - \overleftarrow{f}(S)$

$\leq \sum_{u \in S} \sum_{v \in T} f(u,v)$ ∵ if $(u,v) \notin E$; $f(u,v) = 0$

$\leq \sum \sum c(u,v)$

$= cap(S,T)$.

→ Residual Graphs:-

* To find the "maximum" flow; we first select a path & push the max. possible flow into it. To increase |f| even further; we might have to "undo" flow along some edge.

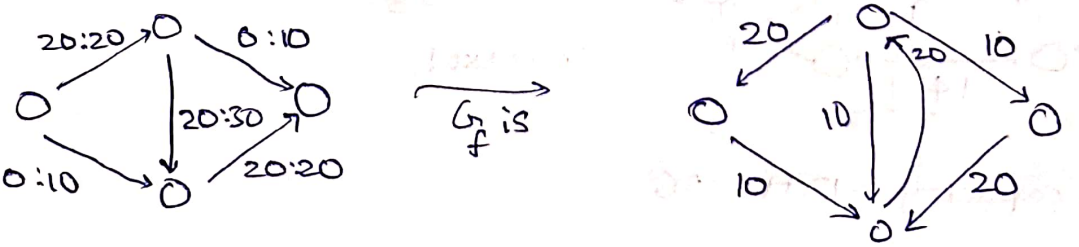
* In order to formalise the idea of "undoing" a flow; we'll introduce residual graph.

Definition:-

graph G_f with capacity function $c: E \to \mathbb{N}$ & flow f; a residual graph of G wrt 'f'; as G_f is:

- The vertices of G_f are same as G.
- if edge e is in G_f such that $f(e) < c(e)$, then G_f has edge e with capacity $c(e) - f(e)$. forward edge.
- if $e = (u,v)$ & $f(e) > 0$; then we add an edge (v,u) in G_f with capacity $f(e)$. Backward edge.

Eg:



these capacities on edges:- residual capacity.

* Augmenting Paths:-

- let Π be any path from s to t in G_f.
- $\theta(\Pi, f)$ denote the smallest residual capacity along Π in G_f.

the subroutine:- Aug(Π,f):-

```
b ← θ(Π,f)
for every e=(u,v) in Π :-
    if e is forward edge then
        f(e) = f(e)+b
    else
        f(v,u) = f(v,u)-b
end for
```

→ Ford-Fulkerson Algorithm:-

For each $e \in E$, set $f(e) \leftarrow 0$

Compute G_f

while There is an s to t path in G_f do

    $f' \leftarrow Aug(\Pi, f)$

    $f \leftarrow f'$

    compute G_f

end while

output f.

→ we need to argue on <our assumption; capacities are integral>.

1) Termination:-

in every iteration; $|f|$ increases by atleast 1.

& $|f| \leq C_{max}$.

Hence terminated!.

---

2) Time analysis:-

Bounded by $O(C_{max} \cdot |E|)$

for loop → atmost C_{max} times

each loop → $O(m)$ + $O(m+n)$ + $O(n)$ ~ $O(|E|)$

($O(m)$ → build G_f; $O(m+n)$ → find path; $O(n)$ → Augmenting)

3) Correctness:-

after terminating;

let say $A \leftarrow \{v \mid v \in V \text{ \& } s,v \text{ are connected in } G_f\}$, $B \leftarrow V \setminus A$.

no forward edge from (A) to (B) in G_f. (lemma → weak duality.)

∴ in G; $E_{A \to B}$ is saturated; $E_{B \to A}$ is avoided. $\{|f| = C(A,B)$ indeed, max value.

---

* At every intermediate stage in our algo; flow values are integral

* If f is our flow, Π is a s-t path, then $|f'| = |f| + \Theta(\Pi, f)$

* $C_{max} = \sum_{v \in V, (s,v) \in E} c(s,v)$. Then maximum flow $\leq C_{max}$.

---

all capacities,

outgoing from start node.

- other issues worth discussing:-

1) is it necessary to choose the right path?

yea. can avoid Cmax iterations, if we take big leaps.

2) max flow problem & min-cut problem. whats the relation.

The (A,B) which we obtained finally is min-cut.

$$\langle |f| \rangle_{max} = cap(A,B) \checkmark$$

3) is it necessary for capacities to be integral?

rational also okay...

$\mathbb{R}$ not okay! since, algo might not terminate.

Applications:-

1) edge disjoint path problem:-

Given: an undirected graph $G = (V, E)$ and $s, t \in V$.

Find: max. no. of edge disjoint paths from s to t.

> Assign capacities of 1 to each edge and done!

2) Network connectivity graph:-

let $G = (V, E)$ is a directed graph. Let $s, t \in V$.

- a subset of the links; $F \subseteq E$ disconnects s from t; if removal of F from the graph disconnects t from s.

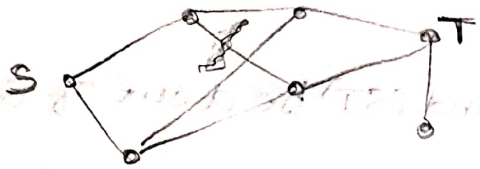
Find: minimum sized set $F \subseteq E$ that disconnects s and t.

→ Menger's Theorem:-

The maximum no. of edge disjoint paths in G = the min. no. of edges to be removed to disconnect t from s.

min. edge removal can be shown as a cut

min.



Proof:

i) we show

min edges ≥ max disjoint path.

ii) we give an example of min-cut as min-edges.

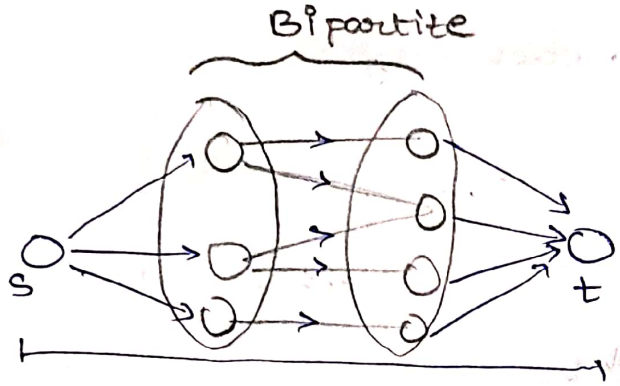
∴ min-edges = max. disjoint.

3) The bipartite max-matching problem :-

Input: an undirected graph $G = (V = (X \cup Y), E)$

Give: largest matching $M \subseteq E$.

Sol: modulate this into a flow network G' :-



flow network.
all edge capacities = 1.

Proof for correctness :-

let flows be integral. Hence $f(\cdot) = 0$ or 1.

in general,
even if c(e) is integral
f(e) need not!
our ford-fulkerson gives
integral flows.

Running time
$= O(|X| \cdot |E|)$.
$= O(|V| \cdot |E|)$

every matching $\Leftrightarrow$ every integral flow instance.

max matching $\Leftrightarrow$ max-flow.

Flow across cut : Lemma :-

let f' be a flow network in G. and (S,T) be a cut of G.

Net flow across the cut; i.e. flow towards T − flow into S is equal to $|f|$.

$$|f| = \overrightarrow{f}(S) - \overleftarrow{f}(S)$$

4) Baseball elimination problem:-

Input:- A set of teams – X.

for each $x \in X$, the number of wins so far, w_x.

for each pair of teams $x, y \in X$; the number of games yet to be played – g_{xy}

a team z; whose fate is needed.

Check:- will z be eliminated? {or "lose" inevitably} draw is fine. But don't lose.

* Try to rephrase as max-flow problem.

step-1:- take it that z wins all its upcoming matches

step-2:- check whether the remaining matches can be played such that no team has more wins that team z.

gives a hint on edge capacities.

whether max flow = Σ matches remaining

> we will draw a flow network G_z.

$$g_z^* = \sum_{x,y \in X \setminus \{z\}} g_{xy}$$

> we show that if the max-flow in G_z is strictly less than g_z^* iff z is eliminated.

cuz; flow = g_z^* with z still present is not possible.

> we argue that construction of g_z takes polynomial time & overall baseball elimination can be solved in poly. time.

* $m = w_z + \sum_{x \in X - \{z\}} g_{zx}$] the best score of 'z' finally.

construction of G_z :-

by defining vertices V; edges E; capacities $c : E \to Z$

- let $X' = X \setminus \{z\}$

1) now $V = \{s, t\} \cup \{u_{xy} \mid x, y \in X'\} \cup \{u_x \mid x \in X'\}$

2) $E = E_{source} \cup E_{mid} \cup E_{sink}$, where

$$E_{source} = \{(s, u_{xy}) \mid x, y \in X'\}$$

$$E_{mid} = \{(u_{xy}, u_x), (u_{xy}, u_y) \mid x, y \in X'\}$$

$$E_{sink} = \{(u_x, t) \mid x \in X'\}.$$

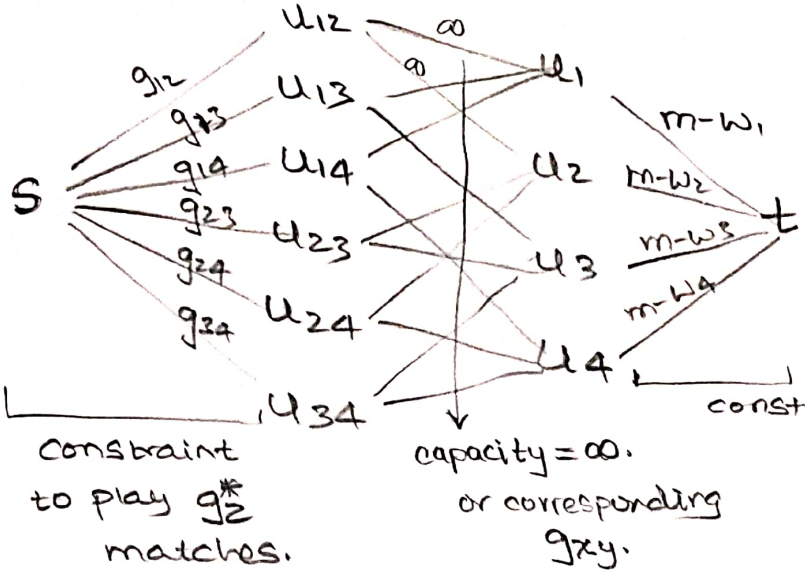
3) capacities as

$$c(e) = \begin{cases} g_{xy} & \text{if } e = (s, u_{xy}) \\ \infty & \text{if } e \in E_{mid} \\ m - w_x & \text{if } e = (u_x, t) \end{cases}$$

its like; we are constraing the other teams to have final score less than z's final score (i.e. m) & say they have to play total of g_z^* matches.

* if max-flow $= g_z^*$; then its POSSIBLE to have total g_z^* matches & z being the highest scorer.

if max-flow $< g_z^*$; after g_z^* matches, its not possible for z to be on top. Its Eliminated.



$z = 5$.

$m = w_5 + g_{15} + g_{25} + g_{35} + g_{45}$.

Lemma:-

Suppose team-z is eliminated. Then the following two things hold

1) z can finish with atmost wins = $m = w_z + \sum_{x \in X \setminus \{z\}} g_{x,z}$.

2) There exists a bunch of teams $Q \subseteq X \setminus \{z\}$ such that

$$\sum_{x \in Q} w_x + \sum_{x,y \in Q} g_{x,y} > |Q| \cdot m$$

↳ mam gave $\geq |Q| \cdot m$.

by basis of pigeon-hole-principle,

one of the teams got to have more wins than z.

Module3:- NP hardness and reductions:-

* In module 1 and module 2; we elegantly solved many problems.

Are all problems efficiently solvable?

The answer is WE DONOT KNOW. Maybe we're just still not there.

→ There are problems which are believed to be inefficient solutions.

Eg: Scheduling Jobs $J_1, J_2 \ldots J_n$ on two processors.
durations are $d_1\ d_2 \ldots d_n$. I need least time!.

Sol) no efficient solution YET.
every greedy heuristic has counter-example which doesn't result output as the optimal one.

→ P and NP:-

* A problem Π is said to be in the class of P, if there exist an algorithm A such that for any input x, A finds the correct solution in time poly(|x|).

Eg: searching, sorting
Interval scheduling
Integer multiplication, GCD calc^n
String related Optimizations
*Max-flow, min-cut
Primality testing, perfect matching in general graphs.

* for class NP, lets start with some examples

1) 3-colorability problem :-

Given: $G = (V, E)$

Check: Can V be coloured with 3 colors such that all $e = (u, v) \in E$, color of u is not same as v.

HOW?

Brute-force algorithm $\longrightarrow O(3^n)$.

too loong yaaar!

* Suppose, someone provides a coloring $c : V \longrightarrow \{R, G, B\}$.

write a subroutine to check whether it is valid or not.

```
Test-colouring(G, c)
    For each e ∈ E; check if c(u) ≠ c(v)
    If all checks suceed, output 'Yes'
    Else output 'No'.
```

* If colorable, there exists a coloring C that makes Subroutine output 'Yes'

* If not colorable, then for all colorings C, output 'NO'.

2) K-clique

Same sub-routine present.

3) Satisfiability problem

Same sub-routine present.

Definition of NP:-

A problem Π is said to be in NP class if there is a polynomial time algorithm T and the following conditions hold:-

> If input x is a positive instance of Π, then there is a polynomial length proof y such that on input x, y outputs 'Yes'.

> If input x is a negative instance of Π, then for all polynomial proof y, $T(x,y)$ outputs 'No'.

* Every problem in P, is also in NP.

< write some facts >

* Types of problems we've seen so far:-

Decision problems:-

Answer : Yes (or) NO.

Eg: SAT, K-clique existance, primality testing.

Search Problems:-

Find a solution

Eg: ~~finding LCS~~, Finding K-clique...
↳ under optimisation.

Optimisation problems:-

Find the max (min) solution

Eg: max-flow, maximum perfect matching.

1) Decision version of search problem:-
↳ does soln exist? find soln.

Find a path → does there exist a path

Find K-clique → does there exist K-clique.

"If you can solve a search problem, you can solve its "decision" version too!"

2) Decision version of optimisation problem:-
↳ does solution exist with z as atleast 10. (as atmost 20).
↳ minimize (maxi.) a value z

Find Shortestpath → does their exist path of length atmost 20.

"If we can solve optimisation problem, we can solve its decision variant".

→ Polynomial time reduction & NP-completeness:-
(or) NP-hardness

* We will explore the space of np-hard problems, in order to develope a mathematical theory on them.

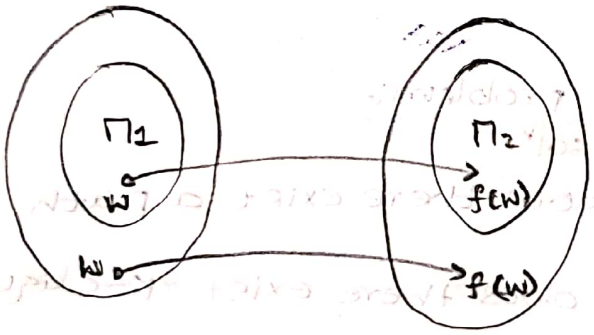
We'll try to understand the relative strength of hard problems, and say "problem Π is atleast as hard as Π^*"!

- We'll use reductions.

Definition:-

A problem Π_1 is said to be polynomial time reducible to another problem Π_2, denote as $\Pi_1 \leq_m \Pi_2$, if the exists a polynomial time computable function f, such that

for all inputs w; $w \in \Pi_1 \Leftrightarrow f(w) \in \Pi_2$.



even P ∈ NP. But not NP-hard.
NOT SAME!

Def^n:- NP-hardness

A problem Π is said to be NP-hard, if for every problem $\Pi' \in NP$, there is a polynomial time reduction, such that $\Pi' \leq_m \Pi$.

Def^n:- NP-completeness

A problem Π is said to be NP-complete if

- Π is in NP
- Π is NP-hard.

] not same.

NP-hard problems are like a family, each reducible to everyother.

* Sorting problem; is in NP. But neither NP-hard; nor NP-complete.

Def^n:- Cook-Levin theorem

SAT is NP-complete.

<visit reference book>

→ Coping with NP-hardness:-

* We can't solve efficiently. But we need to solve somehow na... maybe give "good-enough" solutions.

1. Heuristics :- No guarantees, but works in practise (may)

2. Special cases:- Algos for special inputs.

3. Better-than-brute force Algos:- Like clause learning... DPLL...

4. Approximation algorithms :-

For optimization problems:-

maximization - Finding solution, not less than OPT/c. $c \geq 1$.

minimization - Finding solution, not more than OPT.c. $c \geq 1$.

c-approximation algorithms.

(ideally, should run in polytime)

Eg: we'll see now...

Eg1:- scheduling problem

Given: processors $P_1 P_2 \cdots P_m$ and jobs $j_1 j_2 \cdots j_n$ with durations $d_1 d_2 \cdots d_n$.

find: schedule for these jobs on 'm' processor, that minimizes total completion time.

NP-hard. even for $m=2$. :(

Sol) A_i be the jobs on P_i; $i \in [m]$

$$T_i = \sum_{j \in A_i} d_j.$$

completion time $= \max_{i \in [m]} T_i$

Greedy algo:-

> arrange jobs in arbitrary order

> schedule j^{th} job on machine with smallest load so far.

> let T be the completion time now.

> T^* be the OPT completion time.

we'll show $T \leq 2T^*$. Without knowing what T^* is!

- general strategy:-

1. we'll formulate some quantity T' such that $T' \leq T^*$
2. & we'll show $T \leq 2T'$.

so; $T \leq 2T^*$ will be proved.

* in this case,

$$T^* \geq \frac{1}{m} \left(\sum_{i \in [n]} d_i \right) \quad (A) \qquad \& \text{ also } \quad T^* \geq d_i \ \forall i \in [n] \quad (B)$$

Proof that greedy achieves 2-approximation:-

say i^{th} processor is such that $T^* = T_i$

& say j be the last job on P_i.

$\Rightarrow$ when job j was being alloted; each T_k had value $T_i - d_j$ (atleast)

$$\Rightarrow \sum_{i \in [m]} T_i \geq m (T_i - d_j)$$

$$\therefore \quad T_i - d_j \leq \frac{1}{m} (\Sigma T_i) \leq T^* \qquad (A)$$

also $d_j \leq T^*$ (B)

$\therefore T_i \leq 2T^*$ — 2-approximation.

$\therefore T_{greedy} \leq 2 \cdot T_{OPT}$

poly time algo. — NP-hard problem.

Funfact:-

instead of arbitrary order, if i use sorted array of durations in above greedy algo;

then I can show $T_{greedy} \leq (1.5) T_{opt}$.

Eg:- Smallest vertex cover problem:-

Given: undirected graph $G = (V, E)$

Find: Smallest sized $C \subseteq V$, such that C is a vertex cover.

TL;DR: in CS207, we showed that size of maximal matching is less than size of smallest vertex cover.

$$|M| \leq |V| \leq 2 \cdot |M|$$

$|M|$: any maximal matching; $|V|$: Smallest vertex cover.

Greedy Algo:-

Find a maximal matching M.

E.g. say, our vertex cover V = Set of both ends of edgs in M.

this will indeed be a cover; otherwise M is not maximal.

$\therefore |V| = 2 \cdot |M|$

E.g. $|M| \leq |V_{opt}|$ in CS207. E.g. easy to argue. "Just imagine"

$\therefore |V| \leq 2 \cdot |V_{opt}|$ ✓