NP-Completeness:

- Study this (NP-Completeness) notion & will see how to use this notion effectively.

Motivation:

Imagine that you finished your study, you have gone for interview, you got your dream job, you are really happy, company pays well, you are very happy because you have to deal with design part, and your boss is also happy with you.
He want to give you challenging project.

First Project you get: scheduling jobs on computer

inputs: Set of Jobs & each job has a size associated with it.

Jobs: $j_1, j_2 \cdots j_n$

Sizes: $s_1, s_2, \cdots, s_n$ → positive integer

Two processors: P_1 & P_2 Both are identical

Your Task: To schedule these jobs on these two processors.

Task:- your boss wants to automate this process

You have to write a program, which takes input these sizes $S_1, S_2 \ldots S_n$ (array of size n) & schedule these jobs on processor P_1 & P_2.

→ Now the scheduling should be such that the last job finishes fastest.

Suppose I schedule odd of them on P_1 the time taken will be $S_1 + S_2 + \ldots + S_n$, Lets say that job of size S_i takes same time S_i to finish.

→ Also assume that there is no preemption, once you start the job, it has to run to completion.

→ If I run on one processor then time take is $S_1 + S_2 + \ldots S_n$, all jobs will be finished.

→ Suppose I schedule even jobs on P_2 & odd jobs on P_1.

P_1	P_2	
S_1	S_2	Let us assume
S_3	S_4	7 jobs.
S_5	S_6	
S_7	⋮	

Time taken on $P_1 = S_1 + S_3 + S_5 + S_7$

Time on $P_2 = S_2 + S_4 + S_6$

Assume: jobs scheduled at P_2 are very large jobs. and at P_1, jobs are small.

P_1	P_2
Size of job	Size of job.
1 – S_1	S_2 – 100
1 – S_3	S_4 – 100
1 – S_5	S_6 – 100
1 – S_7	

then time at which last job completes is 300. Unit of time.

→ jobs at P_1 finishes in 4 unit of time & P_2 takes 300 unit of time.

→ This we don't want

First Algo:

→ You take the jobs in order

$S_1, S_2, S_3 \cdots$

P_1	P_2
S_1	S_2
	S_3 (if $S_1 > S_3$)
	S_4 (if $S_2 + S_3 > S_1$)

schedule S_3 at process which is least lightly loaded.

→ This is the first algo, you come up. You code this & your boss is happy but next stage boss tells you that this would not work.

Based on the input you produce the schedule which takes some time T.
While boss produces the schedule which takes less time T.

Ex: Given an example to show that this algo. does not produce an optimum schedule.

⇒ What is wrong in the algo?

→ If jobs are mixed up, i.e size is mixed up arbitrary, may be the reason that algo. doesnot work.

Next attempt ⇒

(i) Sort the jobs by size.
$s_1 < s_2 < s_3 - - - -$

& then run the same algo.

P_1	P_2	
s_1	s_2	
.	.	
.	.	this seems to work. But again after some days, your boss is not happy with algo.
.	.	
.	.	

Ex:- there are five jobs & sizes are given.

2, 2, 2, 3, 3 ← job sizes

P_1	P_2
2	2
2	3
3 (4 < 5)	
7 unit	5 unit

Total time
You schedule
takes

→ you can see that this is not optimum

P_1	P_2
2	3
2	3
2	6 unit
6 unit	

every job is finished in 6 unit of time.

→ It is observed that this algo does not work. Now what to do?

→ May be come up with another heuristic which beat this example but I can tell you that it is extremely difficult, may be ~~impossible~~ impossible to come up with the smart heuristic like this which does well for all inputs.

→ So you can try various heuristic, most of them will not work. You will be really unhappy as boss is threatening you.

What you can do: You want algo that work always.

Brute Force algo:

Which takes every subset of the job.

$S_1, S_2, \ldots, S_n$

Take all subsets S

Compute size(S), size($\bar{S}$) → pass on P_1 & $\bar{S}$ on P_2

→ $\bar{S}$ is the complement of S & Choose the max of these two.

→ do this every possible of subset and pick the minimum

→ You can easily see that this will always give the right answer because you will look at all possible ways of scheduling these jobs on two processors, and one of them has to be the minimum.

→ This is fine, your algo will work for all inputs.

→ Let us your boss fix an input with 1000 nos

→ 1000 jobs have to be scheduled on these 2 processors.

→ Now you start running the algo, 5 minutes passes past, an hour passes, two hours, 10 hours, one day the algo does not stop.
& your boss is worried.

⇒ this is not going to be stopped after many years. that is the problem with brute force algo.

⇒ Let us do the quick calculation, that how much time your algo will take on input of size 1000.

1000 jobs

→ no. of subsets of jobs: 2^{1000}

→ Let us assume that in one instruction the computer can process one of these subsets.

Let us see how much time the computer can take to process all of these subsets.

→ Now the fastest computer runs
Let me make a estimate:
10^{20} instructions/sec. on the fastest computer.

→ How many computers do you think in the world?
10^{20} computers in the world.

→ put all these computers in the world to solve/ run this brute force algo.

putting these together:

10^{40} instructions/sec using all computers.

→ then how many instructions do you think, one can do in year?

$10^{40} \times \underbrace{60}_{\text{minute}} \times \underbrace{60}_{\text{hour}} \times \underbrace{24}_{\text{day}} \times \underbrace{365}_{\text{year}}$ instructions/year.

$10^2 \quad 10^2 \quad 10^2 \quad 10^3$

$\Rightarrow 10^{50}$ insts./year

2^{1000} instructions

Using all computes in the world you can ~~do~~ execute 10^{58} insts/year.

$2^{4*50} \sim 2^{200}$ insts/year

No. of years required:

$2^{1000-200} \sim 2^{800}$ years

↓

so lot of years to finish this algo, this certainly we don't want to do.

→ if your boss figured out this then certainly you will be fired.

So what you will do?

→ this is the motivation to study the NP completeness

→ Final objective is

⇒ To show that this problem is among hard problems in the world.

→ Hard means, no one else will be able to find the solution to this problem.

→ Before we commonly look at NP-completeness, we need to see few other notions.

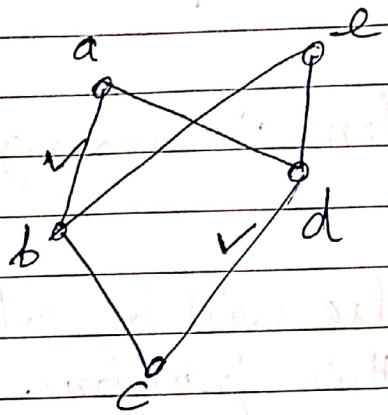
① Reduction:

→ Suppose you have a library, where a large no. of problems are very well coded by very good programmer.

→ Now, a new thing you want to code, it will be very nice if you can use subroutines of the library in your code rather than reinvent and trying by yourself. This is all that reduction. How to efficiently use the code for other algo to generate new algo.

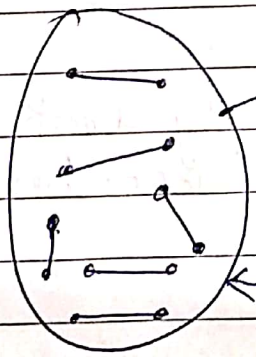
→ Let us look an example: <this is not a very easy Ex.>

→ Matching in a graph $G = (V, E)$ is a subset, M, of E such that no two edges in M have the same end points.



Tick edges are the matching, they do not share their end points.

ab, ad → not a matching



edges will look like this, they do not share ends points.

matching

→ Perfect matching is a matching M such that all vertices are end points of exactly one edge in M.

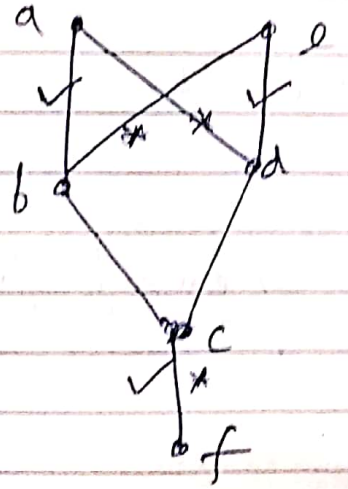
so size of the perfect matching is just half of the size of vertex set,

then $|M| = |V|/2$

→ Do you thing the graph has perfect matching, the answer is no, there are five vertices, so if I take the matching with two edges I can cover four of these vertices, not cover five. I cannot take three edges because two of these edges will share end points.

So this graph does not has the perfect matching.

→ For instance, if I change this graph slightly



✓ : for perfect matching

* : another " "

→ Two Problems:

① ⇒ Problem Perfect Matching:

Input: Graph G.
Question: Does G have a perfect matching?

② ⇒ Problem max matching;
Input; Graph G
output; Matching of max. size

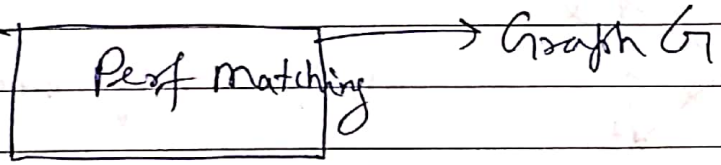
→ Objective:

Given an algo for Perfect matching.

Obj(1) You want to design for max matching.

Intermediate Obj:

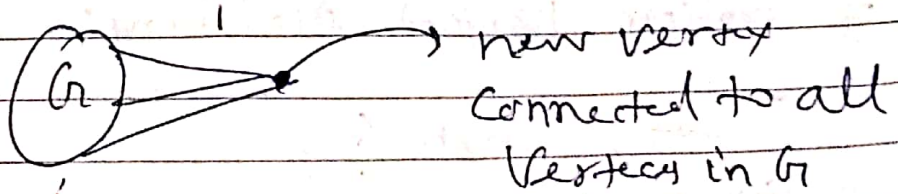
Design an algo to find the size of a max matching.



First Feed in G_1, if G_1 has perfect matching then size is $\frac{|V|}{2}$ ↓ of max matching.

→ Suppose G_1 does not have the perfect matching then what to do?

Step2 → You take



↳ and call this graph G_1

G_1 : G + a vertex connected to all vertices in G

→ If G_1 has a perfect matching that implies that G has a matching of size $\frac{|V|-1}{2}$

→ If G_1 does not have the perfect matching then we create a graph G_2

G_2 : G_1 + a vertex connected to all the vertices in G_1.

⋮

G_3

⋮

G_4

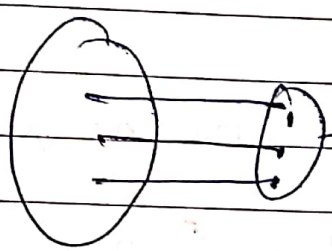
Generic step:

G_i : G_{i-1} + a vertex connected to all the vertices in G_{i-1}

Let us say the first index K, where the perfect matching algo says yes.

Which means:

Assume that
G_{K-1} does not have the perfect matching
&
G_K has a perfect matching.



G_K

$\exists$ a matching in G_K
size $= \frac{n-K}{2}$

So,
$G_1, G_2, G_3 \ldots$
G_{K-1} does not have a perfect matching.
G_K has a perfect matching.