#P-completeness: intro

Recall the definitions of the classes:

**P:** polynomial-time deterministic algo \((\text{on a TM, or any other high-level computational model})\)

**NP:** non-det. poly-time we can understand this as:
- non-det. guess a solution
- deterministically verify its correctness in poly-time

if we have a non-det. TM for a problem in NP \(\Rightarrow\) one of its paths goes to accepting state

**NP-complete:** hardest in NP, can be used to solve everybody else in NP \((\text{all comput. paths polynomial length})\)

A problem is NP-complete if
1) \(\in\) NP
2) is NP-hard \(\Rightarrow\) every problem in NP can be solved via the problem A

i.e. \(B \leq_p A\) polynomially-reducible to A
Class $\mathbf{\#P}$

A class for counting problems.

Def: given a non-det. TM where all comput. paths take a polynomial-time
then we count the number of accepting paths

$\mathbf{\#P}$ contains counting problems that have such a TM

Alternative view: count all solutions to a given problem, where a solution can be verified by a polynomial-time (det) verifier

Examples:

$\text{KNAPSACK} =$ given a set of $n$ objects with weights $W_i$ and cost $c_i$ and a total weight $W$ and a cost $C$, is it possible to choose objects with weight $\leq W$ and cost $\geq C$?

$\mathbf{\#KNAPSACK} =$ same input, count the number of possible subsets of the objects such that weight of the subset $\leq W$ and cost of the subset $\geq C$

i.e. count $\#$ solutions to the KNAPSACK problem

$\text{SAT} =$ given a formula $\phi$, does there exist an assignment True/False to its variables so that

$\mathbf{\#SAT} =$ same input, count the number of possible satisfiable assignments

$\text{If is True ?}$
Class $\#P$

**Note:**
- Since $\text{SAT} \leq_p \#\text{SAT}$, then $\forall B \in \text{NP}$:
  - $B \leq_p \#\text{SAT}$
- (bec. $B \leq_p \text{SAT}$)
- Not in NP since it is not a decision problem
- But it is NP-hard

$\text{SAT} \leq_p \#\text{SAT}$
We say that a problem $f$ is $\#P$-hard, if every problem in $\#P$ is polynomial-time Turing-reducible to $f$.

Turing-reducible: $g \leq_T f$, i.e. $g$ is T-reducible to $f$ iff we can solve $g$ by having an access to a solver of $f$.

A picture for $P, NP, NP$-complete (assuming $P \neq NP$):

- $P$ is contained in $NP$ and $NP$-complete.
- $NP$-complete problems are contained in $NP$ and $NP$-hard.
- $B \rightarrow A$ means that $B \leq_p A$.

Code-like solver for $g$ (input):

1. Transform input and call the solver for $f$.
2. Get a value, say $a$, and feed another input into the solver for $C$.
4. Eventually produce result from numbers $a, b, \ldots$. 

[Section 2.1]
Examples of #P-complete problems:

- #KNAPSACK
- #SAT
- #TRAV.SALESMAN
- etc.

Note: The following problem is OPEN:
If f is NP-complete, is its counting version #P-complete?
\( \text{it is in } \#P \)
# perfect matchings

Consider perfect matchings in bipartite graphs:

- How fast can we find one perfect matching?
- How fast can we count them all?

- poly-time (Kirchhoff) in planar graphs
- in general (or bipartite) graphs
  \( \Rightarrow \) \( \#P \)-complete

poly-time, e.g. by using network flow algorithms
# perfect matchings

**#BipartitePM**

Input: a bipartite graph $G$

Output: # perfect matchings in $G$

**0/1-Perm**

Input: an nxn 0/1 matrix $A$

Output: the permanent

\[
\text{per } A = \sum_{\sigma \in S_n} \prod_{i=0,1,...,n-1} a_{i,\sigma(i)}
\]

**Thm (Valiant):** 0/1-Perm (and #BipartitePM) is #P-complete.
**# perfect matchings**

Thm (Valiant): 0/1-Perm (and \#BipartitePM) is \#P-complete.

Lemma A: \#Exact3Cover \leq_T \#WBipartiteMatch

Lemma B: \#WBipartiteMatch \leq_T \#WBipartitePM

Lemma C: \#WBipartitePM \leq_T \#BipartitePM

Note: we have to define the problems... we'll define only \#WBipartiteMatch and \#WBipartitePM.
# perfect matchings

#WBipartiteMatch

Input: a bip. graph \( G \) with edge weights from \( \{1,-1,-5/3,1/6\} \)

Output: the “total weight” of the matchings in \( G \), i.e.,

\[
P_{\text{match}}(G) = \sum_{M, \text{a match. in } G} \prod_{e \in M} w(e)
\]

#WBipartitePM

- like #WBipartiteMatch but summing only over perf. match.

Note: #WBipartiteMatch and #WBipartitePM are not in \#P...
Lemma B: \( \text{#WBipartiteMatch} \leq_T \text{#WBipartitePM} \)

I.e., we want to show: Having a function for \( \text{#WBipartitePM} \), we can write a func \( \text{Bob} \) for \( \text{#WBipartiteMatch} \) s.t. our func takes poly-number of steps (not counting the running time of \( \text{Perfect Bob} \)).

Consider a weighted bipartite graph \( G \).

Take \( k \geq 0 \)

E.g. \( k = 2 \)

Connect \( G' \) (green & white)

\( \rightarrow \) a (weighted) bipartite graph (\( RUR' \) and \( BVB' \))

form the partition

Connect \( R' \) to \( B \)

(everybody to everybody)

Connect \( R \) to \( B' \)

all new (white) edges

weight 1

Connect \( B \) to \( B' \)

wants (Bob wants)
to compute

\( \sum \) weight of all matchings of \( G \)

r-k nodes

b-k nodes

b nodes

r nodes

\( \text{Perfect Bob} \) (weighted bipartite)
Consider a weighted bipartite graph $G$.

Take $k \geq 0$,

e.g. $k = 2$

There are $k$ nodes on each side.

A perfect matching $M'$ in $G'$:

1) matches all vertices in $R'$ and $B'$, leaving $k$ vertices in $B$ and $k$ vertices in $R$ to match,

$\Rightarrow$ corresponds to a k-matching in $G$, where weight of $M' = \sum$ weight of $M$ (all new edges have weight 1)

2) if we call $\text{PerfectBob}(G')$ it returns $\sum_{\text{perfect match of } G'}$ weight of $M'$

$= (b-k)! (r-k)! \sum_{k\text{-matching of } G} \text{weight of } M$ given a k-matching $M$ of $G$, we have this many possibilities to extend $M$ into a perfect matching of $G'$

connect $R'$ to $B$ (everybody to everybody)

connect $R$ to $B'$

all new (white) edges weight 1
2) if we call $\text{PerfectBob}(G')$ it returns \[
\sum_{\text{perf. match } M' \text{ of } G'} \text{ weight of } M'
\]
\[= (b-k)! (r-k)! \sum_{\text{k-match } M \text{ of } G} \text{ weight of } M \]
given a k-match $M$ of $G$, we have this many possibilities to extend $M$ into a perfect matching of $G'$

```python
func Bob (weighted bip. graph G)
    1. for
```