The class **P**: polynomial time

- Theorems 1 and 2 illustrate an important distinction.
- On the one hand, we demonstrated at most a square or polynomial difference between the time complexity of problems measured on deterministic single tape and multi-tape Turing machines.
- On the other hand, we showed at most an exponential difference between the time complexity of the problems on deterministic and non-deterministic Turing machines.
- For our purpose, polynomial difference in running time are considered to be small, whereas exponential differences are considered to be large.
- Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful. (For *n=1000*, n^3 = 1 billion (still manageable number), 2^n is much larger than the number of atoms in the universe.)
- All reasonable deterministic computational models are polynomially equivalent. Any one of them can simulate another with only a polynomial increase in running time.
- From here on we focus on aspects of time complexity theory that are unaffected by polynomial difference in running time. We consider such differences to be insignificant and ignore them.
- ***The Question is*** whether a given problem is polynomial or non-polynomial.
- So we came to an important definition in the complexity theory, **P** class.

The class **P**: definition

- ***Definition***: **P** is the lass of languages that are decidable in polynomial time on a deterministic single tape Turing machine. That is

$$P = \bigcup_k TIME(n^k).$$

- The class **P** plays an important role in our theory and is important because
  - **P** is invariant for all models of computation that are polynomially equivalent to the deterministic single tape TM, and
  - **P** roughly corresponds to the class of problems that are realistically solvable on a computer.
- When we analyze an algorithm to show that it runs in polynomial time, we need to do two things
  - First, give a polynomial upper bound (usually in big-O notation) on the number of stages that the algorithm uses when it runs on input of length n.
  - Then, examine the individual stages in the description of the algorithm to be sure that each can be implemented in polynomial time on a reasonable deterministic model.
- When both tasks have been done, we can conclude that it runs in polynomial time because we have demonstrated that it runs for a polynomial number of stages, each of which can be done in polynomial time, and the composition of polynomials is a polynomial.

Examples of problems in **P**

- We had: the problem whether *w* is a member of the language $A = \{0^k 1^k : k \geq 0\}$ is in **P.**
- Fortunately, there are many problems that are in **P.**
- The *PATH* problem is to determine whether a directed path exists from *s* to *t*.

$$PATH(G,s,t) = \{<G,s,t>: G \text{ is a directed graph that has a directed path from } s \text{ to } t\}.$$

Theorem: $PATH \in P.$

- we use *breadth first search* and successively mark all nodes in *G* that are reachable from *s* by directed paths of length 1, then 2, then 3, through *m*=|*V*|.

M = "on *<G,s,t>:* where *G* is a directed graph with nodes *s* and *t*.

1. Place a mark on node *s*.
2. Repeat the following until no additional nodes get marked.
3. Scan all the edges of *G*. If an edge *(a,b)* is found going from marked node *a* to an unmarked node *b*, mark *b*.
4. If *t* is marked, *accept*; otherwise *reject*."

- ***Stages 1 ,4*** are executed only once. **Stage 3** runs at most *m*=|*V*| times because each time except the last it marks an additional node in *G*. Hence, the **total number of stages** is *1+1+m*, giving a polynomial in the size of G.

- ***Stages 1,4*** easily implemented in polynomial time on any reasonable deterministic model. **Stage 3** involves a scan of the input and a test whether certain nodes are marked, which also is easily implemented in polynomial time.

- Hence, *M* is a polynomial time algorithm for *PATH*.

The class **NP**

• For some interesting and useful problems, polynomial time algorithms that solve them aren’t known to exist.

• Why have we been unsuccessful in finding polynomial time algorithms for these problems? We don’t know the answer to this important question.

• Perhaps these problems have, as yet undiscovered, polynomial time algorithms that rest on unknown principles.

• Or possibly some of these problems simply cannot be solved in polynomial time. They may be intrinsically difficult.

• One remarkable discovery concerning this question shows that the complexities of many problems are linked. The discovery of a polynomial time algorithm for one such problem can be used to solve an entire class of problems.

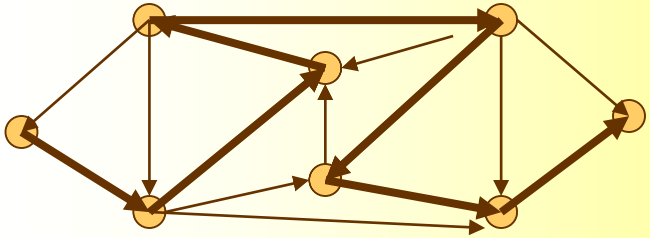
• A ***Hamiltonian path*** in a directed graph *G* is a directed path that goes through each node exactly once. Consider the problem of testing whether a directed graph contains a Hamiltonian path connecting two specified nodes.

HAMPATH={<*G*,*s*,*t*>: *G* is a directed graph with a Hamiltonian path from *s* to *t*}.

• We can easily obtain an exponential time algorithm for the *HAMPATH* problem by brute-force approach which checks all possible permutations of nodes (n!).

• We need only add a check to verify that the potential path is Hamiltonian.

• No one knows whether *HAMPATH* is solvable in polynomial time.



The class **NP**: definition

- Define the **non-deterministic time complexity class**

$$NTIME(t(n)) = \{L : L \text{ is a language decided by an } O(t(n)) \text{ time Non-Deterministic Turing machine}\}.$$

- ***Def:*** **NP** is the class of languages that are decidable in polynomial time on a non-deterministic Turing machine. That is

$$NP = \bigcup_k NTIME(n^k).$$

- The class **NP** is insensitive to the choice of reasonable non-deterministic computation model because all such models are polynomially equivalent.

Theorem: $HAMPATH \in NP.$

- The following is a non-deterministic Turing Machine (NTM) that decides the *HAMPATH* problem in non-deterministic polynomial time (we defined the time of a non-deterministic machine to be the time used by the longest computation branch).

N = "on *<G,s,t>:* where *G* is a directed graph with nodes *s* and *t.*

1. Write a list of *m* numbers $p_1, p_2, \ldots, p_m,$ where *m* is the number of nodes in *G*. Each number in the list is non-deterministically selected to be between *1* and *m*.
2. Check for repetitions in the list. If any are found, *reject.*
3. Check whether $s = p_1$ *and* $t = p_m.$ If either fail, *reject*.
4. For each *i* between *1* and *m-1*, check whether (p_i, p_{i+1}) is an edge of *G*. If any are not, *reject*. Otherwise, *accept*."

- Clearly, this algorithms runs in non-deterministic polynomial time since all stages run in polynomial time.

Polynomial Time Verifiers

- The *HAMPATH* problem does have a feature called ***polynomial verifiability*** that is important for understanding its complexity.

- Even though we don't know of a fast (i,.e., polynomial time) way to determine whether a graph contains a Hamiltonian path, if such a path were discovered somehow (perhaps using the exponential time algorithm), we could easily convince someone else of its existence, simply by presenting it.

- In other words, *verifying* the existence of a Hamiltonian path may be much easier than *determining* its existence.

- We can give an equivalent definition of the **NP** class using the notion ***verifier.***

- A ***verifier*** for a language *A* is an algorithm *V*, where

  $A=\{w: V \text{ accepts } \langle w,c\rangle \text{ for some string } c\}$.

- A verifier uses additional information, represented by the symbol *c* in definition. This information is called a ***certificate***, or ***proof***, of membership in *A*.

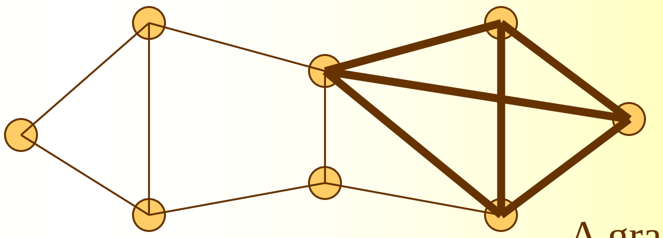
- Example: $\langle G,s,t\rangle$ belongs to *HAMPATH* if for some path *p*, *V* accepts $\langle\langle G,s,t\rangle,p\rangle$ (that is, *V* says "yes, *p* is a Hamiltonian path from *s* to *t* of *G*). For the *HAMPATH* problem, a certificate for a string $<G,s,t>\in HAMPATH$ simply is the Hamiltonian path *p* from *s* to *t*.
- A ***polynomial time verifier*** is a verifier that runs in polynomial time in the length of *w*.
- A language *A* is ***polynomially verifiable*** if it has a polynomial time verifier.
- ***Def:*** **NP** is the class of languages that have polynomial time verifiers.

- **The verifier can check in polynomial time that the input is in the language when it is given the certificate.**

CLIQUE is in **NP**

- A ***clique*** in an undirected graph *G* is a subgraph, wherein every two nodes are connected by an edge. A ***k-clique*** is a clique that contains *k* nodes.
- The ***clique problem*** is to determine whether a graph contains a clique of a specific size.

CLIQUE={<*G*,*k*>: *G* is an undirected graph with a *k*-clique}.



A graph with 4-clique.

Theorem: $CLIQUE \in NP$.

- ***Proof:*** The following is a verifier *V* for *CLIQUE.*

V = "on input <<*G*,*k*>,*c*>*:*

1. Test whether *c* is a set of *k* nodes in *G*.
2. Test whether *G* contains all edges connecting nodes in *c*.
3. If both pass, *accept*; otherwise, *reject*."

- ***Alternative proof:*** If you prefer to think of **NP** in terms of non-deterministic polynomial Turing machine …

N = "on <*G*,*k*>*:* where *G* is an undirected graph, *k* is an integer.

1. Non-deterministically select a subset *c* of *k* nodes in *G*.
2. Test whether *G* contains all edges connecting nodes in *c*.
3. If yes, *accept*; otherwise, *reject*."

SUBSET-SUM is in **NP**

- We have a collection of numbers, $x_1, x_2, ..., x_k,$ and a target number *t*. We want to determine whether the collection contains a subcollection that adds up to *t*.

$$SUBSET-SUM = \{<S,t>: S = \{x_1, x_2, ..., x_k\} \ and$$
$$for \ \ some \ \{y_1, y_2, ..., y_l\} \subseteq \{x_1, x_2, ..., x_k\}, \ we \ \ have \ \sum y_i = t\}.$$

- For example *<{4,11,16,21,27},25>* is in *SUBSET-SUM* since 4+21=25.
- Note that $\{x_1, x_2, ..., x_k\}$ *and* $\{y_1, y_2, ..., y_l\}$ are multisets (we allow repetitions).

Theorem: $SUBSET-SUM \in NP.$

- ***Proof:*** The following is a verifier *V* for *SUBSET-SUM*.

> *V* = "on input *<<S,t>,c>:*
> 1. Test whether *c* is a collection of numbers that sum to *t*.
> 2. Test whether *S* contains all the numbers in *c*.
> 3. If both pass, *accept*; otherwise, *reject*."

- ***Alternative proof:*** If you prefer to think of **NP** in terms of non-deterministic polynomial Turing machine …

> *N* = "on *<S,t>:*
> 1. Non-deterministically select a subset *c* of the numbers in *S*.
> 2. Test whether *c* is a collection of numbers that sum to *t*.
> 3. If yes, *accept*; otherwise, *reject*."

The **P** versus **NP** question

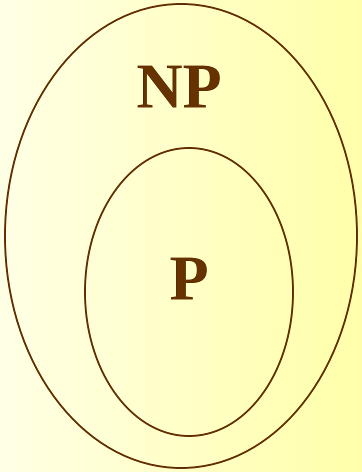
P = the class of languages that are decidable by polynomial time *deterministic* TMs.
NP = the class of languages that are decidable by polynomial time *non-deterministic* TMs.

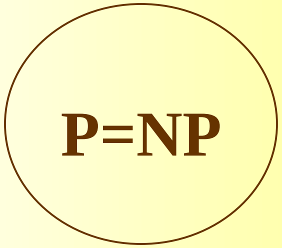
OR EQUIVALENTLY

P = the class of languages where membership can be *decided* quickly (in pol. time).
NP = the class of languages where membership can be *verified* quickly (in pol. time).

- We presented examples of languages, such as *HAMPATH* and *CLIQUE*, that are members of **NP** but that are not known to be in **P**.
- No polynomial time algorithms are known for those problems.
- We are unable to *prove* the existence of a single language in **NP** that is not in **P**.
- The ***question*** of whether **P = NP** is one of the greatest unsolved problems in theoretical computer science.
- Most researchers believe that the two classes are not equal because people have invested enormous effort to find polynomial time algorithms for certain problems in **NP**, without success.
- The best method known for solving problems in **NP** deterministically uses exponential time. In other words, one can show that

$$\bigcup_k NTIME(n^k) = NP \subseteq EXPTIME = \bigcup_k TIME(2^{n^k}).$$





One of these two possibilities is correct.