Basic concepts

Decision Problems:

- When a problem is given, the first thing concern is:
- The problem has a solution or not?

Example-1:

- Consider graph G = (V, E), find if there is a path from vertex 'X' to vertex 'Y' whose length is at most 10.
- Suppose a number is given to you, say 11111111111111111111111 (25 one's), and asks you find it is prime or not.

Observation:

- The above example problems are called decision problems, because the problem has solution (answer), i.e. is either YES or NO.
- Some decision problems can be solved efficiently, using time polynomial to the size of the input of that problem
- Generally 'P' is used to denote the set of all these polynomial-time solvable problems

Example:

- Find the shortest path from vertex 'X' to vertex 'Y' for a graph G.
    - First apply Dijkstra's algorithm and then give the correct answer
    - Hence, this problem is in 'P'
- One more categorization of decision problems is to observe if the problem can be verified in time polynomial toward the input size.

Example:

- For the same graph G, if there is a path from 'X' to 'Y' with length is less than 10, then:
    1. Find the sequence of vertices without repetition in any path from 'X' to 'Y' with length is less than 10.
    2. Verify that the resultant path is in poly-time.
- This type of problem is polynomial-time verifiable.

Examples:

1. Given a graph G (V, E) does the graph contain a Hamiltonian path?
2. Is a given integer ‘x’ a composite number?
3. Given a set of numbers, can be divide them into two groups such that their sum is the same (equal)?

The class NP:

- For some problems, directly we cannot provide solution in polynomial time. Such problems can be represented by ‘NP’.
- A problem is said to be Non-deterministically Polynomial (NP) if we can find a non-determinsitic Turing machine that can solve the problem in a polynomial number of nondeterministic moves.
    - ‘N’ is non-deterministic guessing power of computer
    - ‘P’ is polynomial-time

Examples:

Polynomial problems: searching, sorting and string editing – these problems need the time O(log n), O(n log n) and O(mn) respectively.

Non-Polynomial problems: travelling sales person problem and Knapsack problem -- these problems need time $O(n^2 2^n)$ and $O(2^{n/2})$ respectively.

The two classes of NP problems are:

1. NP- complete problems
2. NP- hard problems

NP-complete problems:

- A decision problem E is NP-complete if and only if every problem in the class NP is polynomial-time reducible to E, i.e.NP-complete can be solved in polynomial time.

Example:

- The decision versions of the Travelling sales person, graph coloring problem and Hamiltonian cycle problem are NP-complete problems.

NP- hard problems:

- Optimization problems whose decision versions are NP-complete are called NP-hard.
- All NP-complete problems are NP-hard but some NP-hard problems are not known to be NP-complete.

Deterministic algorithm:

The class 'P' consists of all problems that can be solved in polynomial time, $O(N^k)$, by deterministic computers.

A *deterministic algorithm* generates the same output for a problem instance every time the algorithm is run

Example: Searching algorithm

/input: an array A[1..N] and an element 'x' that is in the array

//output: the position of the element in array A

```
search(element, A, i)
{
   if (A[i] == x)
       then return i;
   else
       return search(x, A, i+1);
}
```

Let us say, input A = {7, 5, 3, 8, 9, 3} and x = 9 then the search begins from 7, then 5, then 3 and so on until a successful search or element not found.

The complexity of the algorithm is O(n) since the input is not sorted.

Non-deterministic computer:

- A nondeterministic computer has more than one output state and it "chooses" the correct one, is known as "nondeterministic choice".
- A *nondeterministic algorithm* result is not essentially unique.
- A *non-deterministic algorithm* is a two-stage procedure that takes as its input an instance of a decision problem and does the following:

- **Stage 1:** ***Non-deterministic ("Guessing") Stage:*** An arbitrary string 'S' is generated that can be thought of as a candidate solution to the given instance 'x'.
- **Stage 2:** ***Deterministic ("Verification or checking") Stage:*** A deterministic algorithm takes both 'x' and 'S' as its input, and results 'yes' if 'S' represents a solution to instance 'x'. If the checking stage of a nondeterministic algorithm is of polynomial time-complexity, then this algorithm is called an NP (nondeterministic polynomial) algorithm

It makes use of additional functions:

• **choice(S)**: arbitrarily chooses one of the elements of the set S

• **failure()**: signifies unsuccessful computation

• **success()**: signifies successful computation

- A nondeterministic algorithm makes an order of choices for successful termination.
- Two-stage procedure for problem A is:
  - Stage-I: a method makes a guess about the possible solution for problem A.
  - Stage-II: a method checks if the guessed solution is really a solution for problem A.

```
STAGE-1(x, A)
{
    i := random(1 .. n);
    return i;
}
```

```
STAGE-2(i, x, A)
{
    return A[i] == x;
}
```

Example-1: Search algorithm

STAGE-1:

NonDeterministicSearch(x, A)

//input: an array A[1..N] and an element 'x' that is in the array

//output: the position of the element in array A

// informally calls an NDS function, which makes the right choice

{

```
    i := NDS(x, A, 1); // calling function NDS()
     return i;
}
```

The complexity of the algorithm is O(1).

STAGE-2:

```
NDS(x,  A, i)
{
   if (A[i] == x) then
         return i;
   else
         return NDS(x, A, i+1);
}
```

Example-2:

Nondeterministic algorithm to sort the given element of an array A[1..n] is as shown below:

```
Algorithm NDSORT(A, B)
{ // A and B are arrays with size 'n'
   for i := 1 to n do
j := choice(1 . . . n); //selecting correct nondeterministic choice
         if (B[j]!= 0) then
     failure();
B[j] := A[i]; // coping element of A to B
endfor
for i := 1 to n−1 do  //verifying the order of elements
if (B[i] > B[i + 1]) then
      failure(); // if the elements are not in order
```

endfor

write(B);

success();

}

Note:

- We consider only nondeterministic decision algorithms whose output is success or failure.
- Many optimization problems can be stated as decision problems with the property that:
    - The decision problem can be solved in polynomial time if and only if the optimization problem can be solved in polynomial time.

Time Complexity of Nondeterministic Algorithm:

- Time complexity of a nondeterministic algorithm is O(f(n)), for all input size 'n' that result in successful completion and the time required is at most cf(n) where 'c' and 'n_0' are positive constants

For failure or not successful completion the time required is O(1)

Example-1:

Nondeterministic algorithm for Knapsack problem:

```
Algorithm NDKnapsack( p, w, n, m, r, x)
// p-profit, w-weight, n-input size, m-knapsack capacity, r-least profit, x-object
// initialize W and P to zero
{
    for i := 1 to n do
        x[i] := choice(0, 1); // selecting an object (choice of nondeterministic)
        if ( (∑ wi*xi > m) || (∑ pi*xi < r) ) then // knapsack(bag) is empty(not full) or No profit
            failure();
        else
```

success();

}

The time complexity of this algorithm appears to be O(n)+O(m), where, 'm' is the length of the input.

Hence, the algorithm's time complexity is *polynomial* in the size of the input.

Example-2:

Nondeterministic algorithm for Satisfiability (SAT):

- Let $x_1, x_2, \ldots, x_n$ be boolean variables and denotes the negation of x_i
- A formula is an expression of literals combined with boolean operations AND( ) and OR ( ) , where a literal is either a variable or its negation
- A formula is in *conjunctive normal form* (CNF) if it is of the form c_i where, clause $c_i = l_{ij}$
- A formula is in *disjunctive normal form* (DNF) if it is of the form c_i where, clause $c_i = l_{ij}$
- For example, (x_3) (x_1) is in CNF, then the DNF is $(x_1\ x_2)$ (x_3)

The *satisfiability (SAT) problem* is to search out whether or not a formula is true for a few assignment of truth values to the variables.

Nondeterministic Algorithm for SAT:

Algorithm NDSAT(E, n)

//Determine whether the propositional formula 'E' is satisfiable

{

for i := 1 to n do // choose a truth value assignment

x_i = Choice (true, false);

if E($x_1, x_2, \ldots, x_n$) then

success();

else

failure();

}

The time of the algorithm is O(n) + time to deterministically evaluate 'E'.

Therefore, time complexity is O(n) + O(\|E\|).

Thus, this is a polynomial time nondeterministic algorithm.

NP HARD and NP Complete classes

- **P** is set of problems which will be solved in polynomial time.
- **NP** (*nondeterministic polynomial time*) is the set of problems which will be solved in polynomial time.
- NP-complete problems are the "hardest" problems in NP, if any NP-complete problem can be solved in polynomial time, then *all* NP-complete problems can be solved in polynomial time and in fact *every* problem in **NP** can be solved in polynomial.

Reducibility:

If P is *polynomial-time reducible* to Q, we denote this $P \ _p \ Q$, i.e. a problem P can be reduced to another problem Q if any instance of P can be "rephrased" as an instance of Q, the solution to which provides a solution to the instance of P.

NP-Hard problems:

- If all problems R NP are reducible to P, then P is NP-Hard.

NP-Complete problems:

- If P is NP-Hard and P NP then P is NP-Complete.

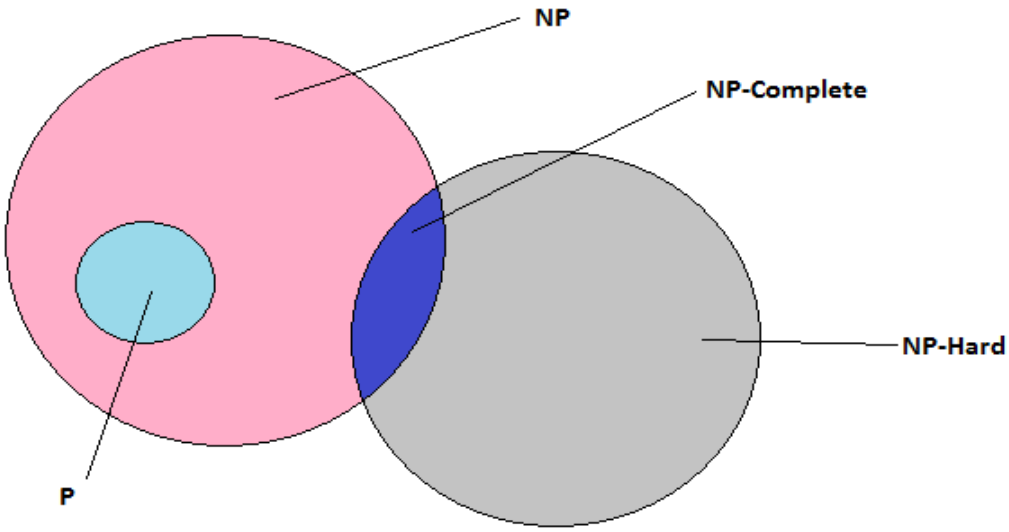

fig. relationship among P, NP, NP-complete and NP-Hard problems

NP-Hard problems are not NP-complete.

Example: HALTING problem:

The decision problem HALTING returns TRUE, if, for a given input 'I' and a given deterministic algorithm 'A'; the algorithm 'A' terminates, otherwise it loops forever. There is no algorithm of any type to solve HALTING.

HALTING problem is NP-hard: *(hint: Reduce an instance of Circuit-satisfiability to an instance of HALTING)*

- The reducing algorithm constructs an instance of HALTING from an instance of CIRCUIT-SAT as follows:
  - Construct an algorithm 'A' whose input is the encoding of a boolean circuit 'B'
  - 'A' tries all possible (2^n, where 'n' is the number of input lines to the circuit) combinations to simulate the circuit, and looks to see if the output is 1 for any of the combinations. If it is, then 'A' halts. If not, then 'A' goes into an infinite loop
- It is clear from the construction that 'A' halts exactly when 'B' is satisfiable, and loops forever exactly when 'B' is not satisfiable.
- In addition, we can perform the construction above in polynomial time:

- The code in algorithm A for the step to generate the 2^n combinations is O(n) long; and if you assume that the number of gates in the boolean circuit is some constant 'k' times the number of inputs, then the code to simulate those gates is also O(n) long

- *Note:* We only need to consider the length of the code in algorithm 'A' and not it's running time, in order to have polynomial time reducibility.

Thus, we conclude by definition of NP-hardness that HALTING is NP-hard

HALTING is not NP-complete:

We know that the HALTING problem is undecidable, i.e., there is no algorithm to solve HALTING. By using this fact, we can prove by contradiction that HALTING is not in NP.

- Let us say a polynomial time verification algorithm for HALTING, given an input string 'x' and an algorithm 'AHNP' such that A(x, y) = 1, with 'x' being an encoding of an instance of HALTING that returns TRUE and 'y' being a certificate that is polynomial length in 'x'

- Given such an AHNP(x, y), we can look at the input string x_{halt} whenever AHNP(x_{halt}, y)=1 and conclude that x_{halt} includes the representation of an algorithm A_{halt} that will halt

The algorithm AHNP can be modified to recognize encodings of all algorithms 'A' that do terminate. But this contradicts the assumption that HALTING is undecidable.

Thus, it is not possible to construct a verification algorithm for HALTING.

Therefore, HALTING is not in NP and by definition of NP-completeness, we conclude that HALTING is not NP-complete.

Hence, **NP-Hard problems are not NP-complete.**

Cook's theorem

- The Cook's theorem (Cook–Levin theorem) states that the boolean satisfiability problem is NP-complete, i.e., any problem in NP can be reduced in polynomial time by a deterministic Turing machine.

- An important consequence of the theorem is that if there exists a deterministic polynomial time algorithm for solving boolean satisfiability, then there exists a deterministic polynomial time algorithm for solving all problems in NP. Crucially, the same follows for any NP complete problem.

Proof:

Let us say a NP decision problem D. And polynomial function P and a Turing machine M for given any instance I (of length 'n') of D, together with a candidate certificate c, will check in time no greater than P(n).

- Let us consider the machine M with q states (*0, 1, 2, .., q*-1) and a tape alphabet (a_1, a_2, .., a_s)
- Initially, the tape is inscribed with the problem instance on the squares *1, 2, .., n* and the putative certificate on the squares *–m, .., -2, -1*
- Square 0 can be assumed to contain a designated separator symbol
- Assume that, the machine halts scanning square 0, and that the symbol in this square at that stage will be a_1 if and only if the candidate certificate is a true certificate
- Here, m <= P(n); since with a problem instance of length 'n' the computation is completed in at most P(n) steps; during this process, the Turing machine head cannot move more than P(n) steps to the left of its starting point

We define some atomic propositions with their proposed interpretations as follows:

1. For *i = 0, 1, .., P(n)* and *j = 0, 1, 2, .., q*-1 the proposition Q_{ij} says that after 'i' computation steps, M is in state 'j'.
2. For *i = 0, 1, .., P(n),* j = *-P(n), …, P(n)* and k =*1, 2, ..,s*; the proposition S_{ijk} says that after 'i' computation steps, square 'j' of the tape contains the symbol a_k.
3. For *i = 0, 1, .., P(n),* j = *-P(n), …, P(n),* the proposition T_{ij} says that after 'i' computation steps, the machine M is scanning square 'j' of the tape.

Next, we define some clauses to describe the computation executed by M:

1. At each computation step, M is in at least one state. For each *i = 0, 1, .., P(n)* we have the clause $Q_{i0} \vee Q_{i1} \vee \ldots Q_{i(q-1)}$ giving *(P(n)+1)q = O(P(n))* literals altogether.
2. At each computation step, M is in at most one state. For each *i = 0, 1, .., P(n)* and for each pair *j, k* of distinct states, we have the clause $(Q_{ij} \wedge Q_{ik})$, giving a total of *q*(q-1)*(P(n) + 1) = O(P(n))* literals altogether.
3. At each step, each tape square contains at least one alphabet symbol. For each *i = 0, 1, .., P(n),* and *-P(n) j P(n)* we have the clause $S_{ij1} \vee S_{ij2} \vee \ldots S_{ijs}$; giving *(P(n)+1)*(2P(n)+1)*s = O(P(n)^2)* literals altogether.
4. In every step, every tape square has at most one alphabet symbol. For every *i = 0, 1, .., P(n),* and *-P(n) j P(n),* and every distinct pair a_k,a_l of symbols we have the clause $(S_{ijk} \wedge S_{ijl})$; giving a total of *(P(n)+1)*(2P(n)+1)*s*(s-1) = O(P(n)^2)* literals altogether.

5. In every step, the tape is scanning at least one square. For each $i = 0, 1, .., P(n)$, we have the clause $T_{i(-P(n))} \vee T_{i(1-P(n))} \vee \ldots T_{i(P(n)-1)} \vee T_{iP(n)}$; giving $(P(n)+1)*(2P(n)+1)=O(P(n)^2)$ literals altogether.

6. In every step, the tape is scanning at most one square. For each $i = 0, 1, .., P(n)$ and each distinct pair j, k of tape squares from -P(n) to P(n), we have the clause $(T_{ij} \wedge T_{ik})$; giving a total of $2P(n)*(2P(n)+1)*(P(n)+1)=O(P(n)^3)$ literals.

7. Initially, the machine is in state 1 scanning square 1. This is expressed by the two clauses Q_{01} and T_{01} giving just two literals.

8. After the first step, at every step the configuration is determined the preceding step by the functions T, U, and D. For every $i = 0, 1, .., P(n), -P(n)\ j\ P(n), k = 0, 1, .., q-1$ and $l = 1, 2, .., s$ we have the clauses

$T_{ij} \wedge Q_{ik} \wedge S_{ijl} \rightarrow Q_{(i+1)T(k, l)}$

$T_{ij} \wedge Q_{ik} \wedge S_{ijl} \rightarrow S_{(i+1)jU(k, l)}$

$T_{ij} \wedge Q_{ik} \wedge S_{ijl} \rightarrow T_{(i+1)(j+D(k, l))}$

$S_{ijk} \rightarrow T_{ij} \vee S_{(i+1)jk}$

The fourth of these clauses ensures that the contents of any tape square ther

than the currently scanned square remains the same.

These clauses contribute a total of $(12s + 3)(P(n) + 1)(2P(n) + 1)q = O(P(n)^2)$ literals.

9. Initially, the string $a_{i1}\ a_{i2} \ldots a_{in}$ defining the problem instance 'I' is inscribed on squares $1, 2, \ldots, n$ of the tape. This is expressed by the 'n' clauses $S_{01i1}, S_{02i2}, \ldots, S_{0nin}$ a total of 'n' literals.

10. By the $P(n)^{th}$ step, the machine has reached the halt state, and is then scanning square0, which contains the symbol a_1. This is expressed by the three clauses $Q_{P(n)0}, S_{P(n)01}$ and $T_{P(n)0}$ giving another 3 literals.

Altogether the number of literals involved in these clauses is $O(P(n)^3)$. Thus, the procedure for setting up these clauses, given the original machine M and the instance 'I' of problem D, can be accomplished in polynomial time.

We must now show that we have succeeded in converting the problem D into SAT.

- Suppose first that 'I' is a positive instance of D

- This means that there is a certificate ‘c’ such that when M is run with inputs ‘c’ and ‘I’ it will halt scanning symbol a_1 on square0
- This means that there is some sequence of symbols that can be placed initially on squares *-P(n), -P(n-1), .., -1* of the tape so that all the clauses above are satisfied

Hence, those clauses constitute a positive instance of SAT.

Conversely, suppose ‘I’ is a negative instance of D. In that case there is no certificate for ‘I’, which means that whatever symbols are placed on squares *-P(n), -P(n-1), .., -1* of the tape, when the computation halts the machine will not be scanning a_1 on square0. This means that the set of clauses above is not satisfiable, and hence constitutes a negative instance of SAT.

- Thus from the instance ‘I’ of problem D is constructed in polynomial time, a set of clauses which constitute a positive instance of SAT if and only ‘I’ is a positive instance of D.
- Hence, the problem D converted into SAT in polynomial time.
- And since D was an arbitrary NP problem it follows that any NP problem can be converted to SAT in polynomial time.