UNIT – 6

NP Hard and NP Complete

Polynomial Time	**Exponential Time**
Linear search-n	0/1 knapsack - 2^n
Binary search-log n	Travelling sp -2^n
Insertion sort- n^2	sum of subsets – 2^n
Merge sort – n log n	Graph colouring – 2^n
Matrix multiplication -n^3	Hamiltonian cycle -2^n

Actually this topic is research topic. The algorithms are categorized into two types:

1. Polynomial time taking algorithms
2. Exponential time taking algorithms

Here we have linear search algorithm which takes O(n) time and faster than that is binary search algorithm which takes O(log n) time , and for sorting algorithm merge sort which takes O(n log n) which takes less time and we need faster algorithms which is faster than this one.

Similarly for these exponential time algorithms we need polynomial time taking algorithms. All these algorithms which taking exponential time we want faster and easy method to solve them in just polynomial time because 2^n, n^n , etc.. are much bigger than polynomial time.

As these exponential algorithms are very time consuming so we need polynomial time for them, this is our requirement.

This is a research area and the person from computer science and mathematics can solve these problems so people have been doing research on this one but there is no particular solution found so far yet for solving these problems in polynomial time.

Then ,when the research work is going fruitless we want something such that whatever the work we done should be useful so these are the guidelines or framework made for doing research on these type of problems i.e., exponential type problems and that framework is NP Hard and NP Complete.

Let us see the basic idea behind that so there are two points on which the entire topic is based on these two points only.

1. When you are unable to solve these exponential problems, that you are unable to get solution in polynomial time algorithm for them at least you do the work such that you try to show the similarities between them so that if one problem is solved the other can also be solved. We will not be doing research work individually on each and every problem like one is working on knapsack problem and the other on travelling sp problem.

Let us combine them collectively and put some effort such that if one problem is solved all the other problems should be solved. So far that we have to show the relationship between them.

"Try to relate the problem either solve it or at least related"

2. When we are unable to write algorithm for them that is deterministic, why don't we write non deterministic algorithm.

Deterministic Algorithm:

Each and every statement how it works we know it clearly. We are sure how they work i.e., we know the working of the algorithm. This type of algorithms are called deterministic algorithms.

Non deterministic Algorithms

Here, we don't know how the algorithm works. Then how can we write the algorithms which doesn't know? While writing an algorithm some of the statements may not be figuring out,how to make them polynomial so leave them blanks and say this is non deterministic and when we know how it has to be filled ,we fill up that statements.

In a non-deterministic algorithm also most of the statements may be deterministic, but some of the statements are nondeterministic

By writing this non-deterministic algorithms we can preserve our research work so that in future somebody else work on this same problem. He/she can make the nondeterministic part as deterministic. This is the main idea in writing non deterministic algorithms.

Let us take an example :

Non deterministic Search Algorithm

```
Algorithm Nsearch (A, n, key)
{
        J=choice();
        If(key=A[i])
        {
                Write(j);
                Success();
        }
        else{
                write(0);
                failure();
        }
}
```

Assume that all these statements (choice(), success(), failure()) takes just one unit of time.

This algorithm is about searching a key from an array of N elements.

While seeing this algorithm we can say that the algorithm takes O(1) time.this is the constant time algorithm for searching and the fastest searching algorithm is binary search that takes O(log n) time. And this algorithm takes O(1) and is really faster , we need this one but it is non deterministic.

In the above algorithm choice() gives the index of key element and we are checking the key element is present at index j ,then search is successful otherwise failure and if key element is not present at index 'j' then key element is not present in the array then search is unsuccessful

Now the question is how this choice() knows that the key element is present at j th index.that's what it is non deterministic.Once we know the position of key element in constant time then we will fill up the choice() line. Choice() line is like a blank statement for us,now once we fill this we have an algorithm of O(1) for searching.

Example: we have an array .

10	8	5	9	4	2

If key=5. Then choice() will give directly index 2.but how we doesn't know

Similarly, we write non deterministic algorithms for exponential problems with this we define two classes

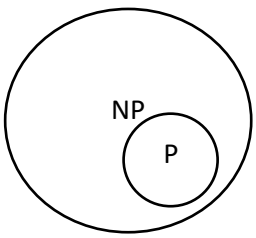
P and NP classes:

P - ' P ' is a set of those deterministic algorithms which are taking polynomial time

Example linear search, binary search, bubble sort etc…

NP -These algorithms are non deterministic but they take polynomial time. We actually write these algorithms for exponential time taking algorithms.

NP- Non deterministic Polynomial time taking algorithms.



' P ' is shown as subset of ' NP ' which means the deterministic algorithms that we know today were the part of non deterministic algorithms.

We completed the first thing that is writing non deterministic polynomial time taking algorithms. How to write is completed

Now the second part if you are unable to solve , atleast show the relationship between them such that if one problem it shoulb be easy for solving the other problems also in polynomial time .we will not work on each problem individually so how to relate them that is the next thing we are going to discuss

So for relating them together we need some problem as a base problem and the base problem is satisfiability problem

satisfiability:-

- In polynomial time, if you unable to solve at least show the relationship between them such that if one problem is solved, then it should be easy for solving other problems also in polynomial time.
- We will not work on each and every problem individually.so for relating them together we need some problem as base problem.
- The base problem is Satisfiability.
- Let x_1, x_2, x3…. xn denotes Boolean variables
- .Let xi denotes the relation of x_i.
- A literal is either a variable or its negation
- .A formula in the prepositional calculus is an expression that can be constructed using literals and the operators and Λ or v.
- A clause is a formula with at least one positive literal.
- The satisfiability problem is to determine if a formula is true for some assignment of truth values to the variables.
- It is easy to obtain a polynomial time non determination algorithm that terminates successfully if and only if a given prepositional formula E (x_1, x_2……x_n) is satiable.
- Such an algorithm could proceed by simply choosing (non deterministically) one of the 2 n possible assignments of truth values to (x_1, x_2…x_n) and verify that E (x_1, x_2…x_n) is true for that assignment.

The satisfiability problem:-

The logical formula:

X_1 v X_2 v X_3

& - x_1

& - x_2

The assignment:

$x_1 \leftarrow$ F, $x_2 \leftarrow$ F, $x_3 \leftarrow$ T

will make the above formula ∀≠true.

(-x_1, -x_2, x_3) represents $x_1 \leftarrow$ F, $x_2 \leftarrow$ F, $x_3 \leftarrow$ T

- If there is at least one assignment which satisfies a formula, then we say that this formula is satisfiable; otherwise, it is unsatisfiable.

- An unsatisfiable formula:

x_1 v x_2

& x1 v –x

& -x_1 v x_2

& -x_1 v -x_2

• Definition of the satisfiability problem:

Given a Boolean formula, determine whether this formula is satisfiable or not.

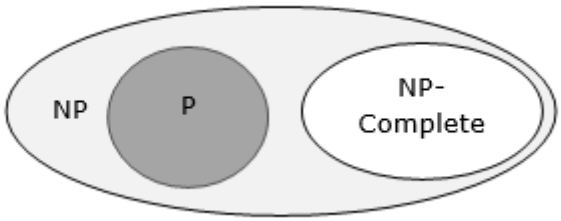
• A literal: x_i or -x_i

• A clause: x_1 v x_2 v -x_3 Ci

• A formula: conjunctive normal form (CNF) C1& C2& … & Cm

NP hard and NP complete Classes:

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness:

A language **B** is ***NP-complete*** if it satisfies two conditions

- **B** is in NP
- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus, ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle*** $\leq_p$ ***TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph $G' = (V, E')$, where

$$E'=\{(i ,j): i , j \in V \text{ and } i \neq j$$

Thus, the cost function is defined as follows −

$$t(i, j)= \{0 \text{ if}(I ,j)\in E$$

$$1 \quad \text{Otherwise,}$$

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in G' as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in G'. Thus, if graph ***G*** has a Hamiltonian cycle, then graph G' has a tour of **0** cost.

Conversely, we assume that G' has a tour h' of cost at most **0**. The cost of edges in E' are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of h' is **0**. We therefore conclude that h' contains only edges in E.

We have thus proven that G has a Hamiltonian cycle, if and only if G' has a tour of cost at most **0**. TSP is NP-complete.

NP-HARD GRAPH AND SCHEDULING PROBLEMS

Some NP-hard Graph Problems:

The strategy to show that a problem L_2 is NP-hard is

I. Pick a problem L_1 already known to be NP-hard.

II. Show how to obtain an instance I^1 of L_2 from any instance I of L_1 such that from the solution of I^1 - We can determine (in polynomial deterministic time) the solution to instance I of L_1.

III. Conclude from (ii) that $L_1 \alpha L_2$.

IV. Conclude from (i), (ii), and the transitivity of α that Satisfiabilityα L1
$L_1 \alpha L_2$
Therefore, Satisfiability L_2
and L_2 is NP-hard

1. Chromatic Number Decision Problem (CNP)

- A coloring of a graph G = (V, E) is a function f: V → {1,2, …, k} ∀ I ϵ V.
- If (U, V) ϵ E then f(u) ≠f(v).
- The CNP is to determine if G has a coloring for a given K.
- Satisfiability with at most three literals per clause α chromatic number problem. Therefore, CNP is NP-hard.

2. Directed Hamiltonian Cycle (DHC): -

- Let G = (V, E) be a directed graph and length n = 1V1
- The DHC is a cycle that goes through every vertex exactly once and then returns to the starting vertex.
- The DHC problem is to determine if G has a directed Hamiltonian Cycle.

Theorem: CNF (Conjunctive Normal Form) satisfiabilityα DHC.
Therefore, DHC is NP-hard.

3. Problem (TSP) :Travelling Salesperson Decision:

• The problem is to determine if a complete directed graph G = (V, E) with edge costs C(u, v) has a tour of cost at most M.

Theorem: Directed Hamiltonian Cycle (DHC) αTSP

• But from problem (2) satisfiabilityα DHC

Therefore, Satisfiability α TSP
and TSP is NP-hard.

NP-HARD SCHEDULING PROBLEMS

Sum of subsets: -

The problem is to determine if A= {a_1, a_2 ……., a_n} (a1, a_2 …….,a_n are positive integers) has a subset S that sums to a given integer M. 2.

Schedule identical processors: -

- Let P_i $1 \leq i \leq m$ be identical processors or machines Pi.
- Let J_i $1 \leq i \leq n$ be n jobs.
- Jobs $J_i$requires t_i processing time.
- A schedule S is an assignment of jobs to processors.
- For each job Ji, S specifies the time intervals and the processors on which this job is to be processed.
- A job cannot be processed by more than one processor at any given time. The problem is to find a minimum finish time non-preemptive schedule.
- The finish time of S is FT(S) = max $\{T_i\}$ $1 \leq i \leq m$.
- Where T_i is the time at which processor Pi finishes processing all jobs (or job segments) assigned to it.

Cook's Theorem:

- The Cook-Levin theorem, also known as Cook's theorem, states that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic Turing machine to the problem whether a Boolean formula is satisfiable.
- The theorem is named after Stephen Cook and Leonid Levin.
- An important consequence of this theorem is that if there exists a deterministic polynomial time algorithm for solving Boolean satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm.

The work shows that Cook's theorem is the origin of the loss of non-determinism in terms of the equivalence of the two definitions of NP, the one defining NP as the class of problems solvable by a nondeterministic Turing machine in polynomial time, and the other defining N P as the class of problems verifiable by a deterministic Turing machine in polynomial time. Therefore, we argue that fundamental difficulties in understanding P versus NP lie firstly at cognitionlevel, then logic level.

Following are the four theorems by Stephen Cook −

Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

- For any $T_Q(k)$ of type **Q**, TQ(k)k√(logk)2TQ(k)k(logk)2 is unbounded
- There is a $T_Q(k)$ of type **Q** such that TQ(k)⩽2k(logk)2TQ(k)⩽2k(logk)2

Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type **Q**, then there is a constant **K**, so **S** can be recognized by a deterministic machine within time $T_Q(K8^n)$.

- First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.
- Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.
- Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be

solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.

- Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

In order to prove this, we require a uniform way of representing NP problems. Remember that what makes a problem NP is the existence of a polynomial-time algorithm more specifically, a Turing machine for checking candidate certificates.

Assume, then, that we are given an NP decision problem D. By the definition of NP, there is a polynomial function P and a Turing machine M which, when given any instance I of D, together with a candidate certificate c, will check in time no greater than P(n), where n is the length of I, whether or not c is a certificate of I

Let us assume that M has q states numbered 0, 1, 2, . . . , q − 1, and a tape alphabet a1, a2, . . . , as. We shall assume that the operation of the machine is governed by the functions T, U, and D. We shall further assume that the initial tape is inscribed with the problem instance on the squares 1, 2, 3, . . . , n, and the putative certificate on the squares −m, . . . , −2, −1. Square zero can be assumed to contain a designated separator symbol. We shall also assume that the machine halts scanning square 0, and that the symbol in this square at that stage will be a1 if and only if the candidate certificate is a true certificate. Note that we must have $m \leq P(n)$. This is because 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 intended interpretations as follows:

1. For i = 0, 1, . . . , P(n) and j = 0, 1, . . . , q − 1, the proposition Qij 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 Sijk says that after i computation steps, square j of the tape contains the symbol ak.

3. i = 0, 1, . . . , P(n) and j = −P(n), . . . , P(n), the proposition Tij 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, . . . , P(n) we have the clause

$$Qi0 \vee Qi1 \vee \cdots \vee Qi(q-1),$$

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, . . . , P(n) and for each pair j, k of distinct states, we have the clause

$$\neg(Qij \wedge Qik),$$

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, . . . , P(n) and −P(n) ≤ j ≤ P(n) we have the clause

$$Sij1 \vee Sij2 \vee \cdots \vee Sijs,$$

giving (P(n) + 1)(2P(n) + 1)s = O(P(n) 2) literals altogether.

4. At each step, each tape square contains at most one alphabet symbol. For each i = 0, . . . , P(n) and −P(n) ≤ j ≤ P(n), and each distinct pair ak, al of symbols we have the clause

$$\neg(Sijk \wedge Sijl),$$

giving a total of (P(n) + 1)(2P(n) + 1)s(s − 1) = O(P(n) 2) literals altogether

5. At each step, the tape is scanning at least one square. For each i = 0, . . . , P(n), we have the clause

$$Ti(-P(n)) \vee Ti(1-P(n)) \vee \cdots \vee Ti(P(n)-1) \vee TiP(n),$$

giving (P(n) + 1)(2P(n) + 1) = O(P(n) 2) literals altogether.

6. At each step, the tape is scanning at most one square. For each i = 0, . . . , P(n), and each distinct pair j, k of tape squares from −P(n) to P(n), we have the clause

$$\neg(Tij \wedge Tik),$$

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

$$Q01,\ T01,$$

giving just two literals.

8. The configuration at each step after the first is determined from the configuration at the previous step by the functions T, U, and D defining the machine M. For each i = 0, . . . , P(n), −P(n) ≤ j ≤ P(n), k = 0, . . . , q − 1, and l = 1, . . . , s, we have the clauses

$$Tij \wedge Qik \wedge Sijl \rightarrow Q(i+1)T(k,l)$$
$$Tij \wedge Qik \wedge Sijl \rightarrow S(i+1)jU(k,l)$$
$$Tij \wedge Qik \wedge Sijl \rightarrow T(i+1)(j+D(k,l))$$
$$Sijk \rightarrow Tij \vee S(i+1)jk$$

The fourth of these clauses ensures that the contents of any tape square other than the currently scanned square remains the same (to see this, note that the given clause is equivalent to the formula Sijk ∧ ¬Tij → S(i+1)jk). These clauses contribute a total of (12s + 3)(P(n) + 1)(2P(n) + 1)q = O(P(n) 2) literals.

9. Initially, the string ai1 ai2 . . . ain defining the problem instance I is inscribed on squares 1, 2, . . . , n of the tape. This is expressed by the n clauses

$$S01i1\ ,\ S02i2\ ,\ \ldots\ ,\ S0nin\ ,$$

a total of n literals.

10. By the P(n)th step, the machine has reached the halt state, and is then scanning square 0, which contains the symbol a1. This is expressed by the three clauses

$$QP(n)0,\ SP(n)01,\ TP(n)0,$$

giving another 3 literals.

Altogether the number of literals involved in these clauses is $O(P(n)^3)$ (in working this out, note that q and s are constants, that is, they depend only on the machine and do not vary with the problem instance; thus they do not contribute to the growth of the the number of literals with increasing problem size, which is what the O notation captures for us). It is thus clear that 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 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, I, it will halt scanning symbol a1 on square 0. This means that there is some sequence of symbols that can be placed initially on squares $-P(n), \ldots, -1$ 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 $-P(n), \ldots, -1$ of the tape, when the computation halts the machine will not be scanning a1 on square 0. 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 we have 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. In other words, we have converted D 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.