P, NP, NP-Hard, NP-Complete

NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved.

P- Polynomial time **solving** . Problems which can be solved in polynomial time, which take time like O(n), $O(n^2)$, $O(n^3)$. Eg: finding maximum element in an array or to check whether a string is palindrome or not. so there are many problems which can be solved in polynomial time.

NP- Non deterministic Polynomial time solving. Problem which can't be solved in polynomial time like TSP(travelling salesman problem) or An easy example of this is subset sum: given a set of numbers, does there exist a subset whose sum is zero?. These problems can’t be solved in polynomial time by deterministic algorithms but they can be solved in polynomial time by nondeterministic algorithms.

but NP problems are **checkable** in polynomial time means that given a solution of a problem , we can check that whether the solution is correct or not in polynomial time.

So NP is the set of decision problems that can be verified in polynomial time.

33.2. NP-Hard and NP-Complete Problems

33.2.1. NP-Hard Problems

- We say that a decision problem P_i is *NP-hard* if *every* problem in **NP** is polynomial-time reducible to P_i.
- In symbols,

  P_i is **NP**-hard if, for every $P_j \in \mathbf{NP}, P_j \xrightarrow{\text{poly}} P_i$.
- Note that this doesn't require P_i to be in **NP**.
- Highly informally, it means that P_i is 'as hard as' all the problems in **NP**.
  - If P_i can be solved in polynomial-time, then so can all problems in **NP**.
  - Equivalently, if any problem in **NP** is ever proved intractable, then P_i must also be intractable.

33.2.2. NP-Complete Problems

- We say that a decision problem P_i is *NP-complete* if
  - it is **NP**-hard and
  - it is also in the class **NP** itself.
- In symbols, P_i is **NP**-complete if P_i is **NP**-hard and $P_i \in \mathbf{NP}$

Examples of algorithms which can be solved by algorithms that take polynomial time and exponential time are:

Polynomial Time	Exponential time
Linear Search – O(n)	Travelling Salesman Problem – $O(n^2 2^n)$
Binary Search –O(logn)	Knapsack Problem – $O(2^{n/2})$
Selection Sort –$O(n^2)$	Sum of Subsets –$O(2^n)$
Merge Sort –O(nlogn)	Graph coloring – $O(nm^n)$ where n is the no. of vertices, m is the no. of colours
Matrix multiplication – $O(n^3)$	Hamiltanion Cycle – $O(2^n)$
Ordered Searching – O(logn)	
Polynomial Evaluation –O(n)(when Horners Rule is used)	
Huffman Coding, Optimal merge patterns –$O(n^2)$	

Many of the problems which do not have polynomial time algorithms are computationally related.

A problem that is NP-complete has the property that it can be solved in polynomial time if and only if all other NP-complete problems can also be solved in polynomial time. A problem is said to be NP-hard if every problem in NP-complete set is reducible to it.

Reducibility- If we can convert one instance of a problem A into problem B (NP problem) then it means that A is reducible to B.

A problem is said to be NP-hard if everything in NP can be transformed in polynomial time into it, and a problem is NP-complete if it is both in NP and NP-hard. The NP-complete problems represent the hardest problems in NP. If any NP-complete problem has a polynomial time algorithm, all problems in NP do. Although a solution to an NP-complete problem can be *verified* "quickly", there is no known way to *find* a solution quickly. That is, the time required to solve the problem using any currently known algorithm increases rapidly as the size of the problem grows.

	Time for $f(n)$ instructions on a 10^9 instr/sec computer						
n	$f(n)=n$	$f(n)=n\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01 μs	.03 μs	.1 μs	1 μs	10 μs	10 s	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	160 μs	2.84 hr	1 ms
30	.03 μs	.15 μs	.9 μs	27 μs	810 μs	6.83 d	1 s
40	.04 μs	.21 μs	1.6 μs	64 μs	2.56 ms	121.36 d	18.3 min
50	.05 μs	.28 μs	2.5 μs	125 μs	6.25 ms	3.1 yr	13 d
100	.1 μs	.66 μs	10 μs	1 ms	100 ms	3171 yr	$4*10^{13}$ yr
1,000	1 μs	9.96 μs	1 ms	1 s	16.67 min	$3.17*10^{18}$ yr	$32*10^{283}$ yr
10,000	10 μs	130 μs	100 ms	16.67 min	115.7 d	$3.17*10^{23}$ yr	
100,000	100 μs	1.66 ms	10 s	11.57 d	3171 yr	$3.17*10^{33}$ yr	
1,000,000	1 ms	19.92 ms	16.67 min	31.71 yr	$3.17*10^{7}$ yr	$3.17*10^{43}$ yr	

Table 1.8 Times on a 1-billion-steps-per-second computer

The above table shows the time taken by different algorithms with different time complexities.

Nondeterministic algorithms

Algorithms whose every operation is **uniquely defined** are called deterministic algorithms. Such algorithms agree with the way programs are executed on a computer. In theory we can remove this restriction on the outcome of every operation. We can allow **algorithms to contain operations whose outcomes are not uniquely defined but are limited to a specified set of possibilities**. The machine executing such operations is allowed to choose any one of these outcomes subject to a termination condition. This leads to the concept of a non deterministic algorithm. To specify such algorithms, three functions are used.

1. Choice(S): arbitrarily chooses one of the elements of set S.

2. Failure() signals an unsuccessful completion

3. Success() signals a successful completion.

The assignment statement x:= Choice(1,n) could result in x being assigned one of the integers in the range [1,n]. There is no rule specifying how this choice is to be made. The Failure() and Success() signals are used to define a computation of the algorithm. **Whenever there is a set of choices that leads to a successful completion, then one such set of choices is always made and the algorithm terminates successfully. A nondeterministic algorithm terminates unsuccessfully if and only if there exists no set of choices leading to a success signal.**

The computing times of Choice, Success and Failure are taken as O(1). A machine capable of executing a nondeterministic algorithm in this way is called a nondeterministic machine (which does not exist in practice).

Nondeterministic algorithm for search

Example 11.1 Consider the problem of searching for an element x in a given set of elements $A[1:n]$, $n \geq 1$. We are required to determine an index j such that $A[j] = x$ or $j = 0$ if x is not in A. A nondeterministic algorithm for this is Algorithm 11.1.

```
1   j := Choice(1, n);
2   if A[j] = x then {write (j); Success();}
3   write (0); Failure();
```

Algorithm 11.1 Nondeterministic search

The above algorithm outputs 0 if and only if there is no j such that A[j]=x; If 'x' is in A, Choice will somehow find its index (how it finds is not known now but some researcher might find it in future). So only if 'x' is not in A the algorithm terminates unsuccessfully.

The time complexity of this nondeterministic algorithm is O(1). It should be observed that if A is unordered, every deterministic search algorithm (like linear search) has a time complexity of $\Omega(n)$

Nondeterministic algorithm for Sorting:

The nondeterministic sorting algorithm for sorting somehow determines the right position where every element should be there after sorting (this is done by the Choice function). The element is then placed in second array B in that position. Finally in line 11 and 12 it is verified if every element in in B is less than its next consecutive element in B if not the algorithm terminates unsuccessfully otherwise the algorithm terminates successfully.

Example 11.2 [Sorting] Let $A[i]$, $1 \leq i \leq n$, be an unsorted array of positive integers. The nondeterministic algorithm NSort(A, n) (Algorithm 11.2) sorts the numbers into nondecreasing order and then outputs them in this order. An auxiliary array $B[1 : n]$ is used for convenience. Line 4 initializes B to zero though any value different from all the $A[i]$ will do. In the **for** loop of lines 5 to 10, each $A[i]$ is assigned to a position in B. Line 7 nondeterministically determines this position. Line 8 ascertains that $B[j]$ has not already been used. Thus, the order of the numbers in B is some permutation of the initial order in A. The **for** loop of lines 11 and 12 verifies that B is sorted in nondecreasing order. A successful completion is achieved if and only if the numbers are output in nondecreasing order. Since there is always a set of choices at line 7 for such an output order, algorithm NSort is a sorting algorithm. Its complexity is $O(n)$. Recall that all deterministic sorting algorithms must have a complexity $\Omega(n \log n)$. □

```
1   Algorithm NSort(A, n)
2   // Sort n positive integers.
3   {
4       for i := 1 to n do B[i] := 0; // Initialize B[ ].
5       for i := 1 to n do
6       {
7           j := Choice(1, n);
8           if B[j] ≠ 0 then Failure();
9           B[j] := A[i];
10      }
11      for i := 1 to n − 1 do  // Verify order.
12          if B[i] > B[i + 1] then Failure();
13      write (B[1 : n]);
14      Success();
15  }
```

Algorithm 11.2 Nondeterministic sorting

Definition: Any problem for which the answer is either zero or one is called a decision problem. An algorithm for a decision problem is termed a decision algorithm. Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an optimization problem. An optimization algorithm is used to solve an optimization problem.

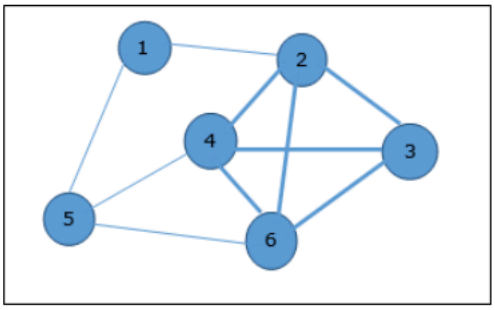
We concentrate on nondeterministic decision algorithms. Although we are restricting ourselves to decision problems, many optimization problems can be recast into decision problems with the property that the decision problem can be solved in polynomial time if and only if the corresponding optimization problem can. In other cases we can at least make a statement that if the decision problem cannot be solved in polynomial time, then the optimization problem cannot either.

Converting Maximum Clique Optimization problem to corresponding Decision Problem

A **maximal complete subgraph** of a graph G= (V,E) is a **clique**. The size of the clique is the number of vertices in it. The max clique problem is an optimization problem that has to **determine the size** of the **largest clique** in G. The corresponding decision problem is to determine whether G has a clique of size atleast 'k' for some given 'k'. The optimization problem can be solved using this decision problem by calling the decision problem for different values of k=n,n-1,n-2,…1.

Example

Take a look at the following graph. Here, the sub-graph containing vertices 2, 3, 4 and 6 forms a complete graph. Hence, this sub-graph is a **clique**. As this is the maximum complete sub-graph of the provided graph, it's a **4-Clique**.




```
1   Algorithm DCK(G, n, k)
2   {
3       S := ∅; // S is an initially empty set.
4       for i := 1 to k do
5       {
6           t := Choice(1, n);
7           if t ∈ S then Failure();
8           S := S ∪ {t} // Add t to set S.
9       }
10      // At this point S contains k distinct vertex indices.
11      for all pairs (i, j) such that i ∈ S, j ∈ S, and i ≠ j do
12          if (i, j) is not an edge of G then Failure();
13      Success();
14  }
```

Algorithm 11.5 Nondeterministic clique pseudocode

Line 4 selects 'k' vertices from the graph. Lines 6 to 9 are used to find the vertices (distinct vertices) that define the clique (again the choice function some how finds these vertices). Lines 11 and 12 are used to check if there is an edge between every pair of vertices in S. If not the algorithm terminates

unsuccessfully otherwise the algorithm terminates successfully. The time complexity of this non deterministic algorithm is determined by lines 11 and 12 in the algorithm which is $O(n^2)$ when adjacency matrix is used.

0/1 Knapsack Optimization Problem

A thief is robbing a store and can carry a maximal weight of *m* into his knapsack. There are *n* items and weight of i^{th} item is w_i and the profit of selecting this item is p_i. What items should the thief take to make maximum profit?

0/1 Knapsack Decision Problem

Example 11.4 [0/1 knapsack] The knapsack decision problem is to determine whether there is a 0/1 assignment of values to x_i, $1 \leq i \leq n$, such that $\sum p_i x_i \geq r$ and $\sum w_i x_i \leq m$. The r is a given number. The p_i's and w_i's are nonnegative numbers. If the knapsack decision problem cannot be solved in deterministic polynomial time, then the optimization problem cannot either.

// n is the number of items
// w is an array containing the weights of n items
// p is an array containing the profits of n items
// m is maximum capacity of knapsack
// r is minimum profit that is to be earned
// x is a Boolean array containing 0s or 1s that specifies which item is selected and which item is not selected

```
1   Algorithm DKP(p, w, n, m, r, x)
2   {
3       W := 0; P := 0;
4       for i := 1 to n do
5       {
6           x[i] := Choice(0, 1);
7           W := W + x[i] * w[i]; P := P + x[i] * p[i];
8       }
9       if ((W > m) or (P < r)) then Failure();
10      else Success();
11  }
```

Algorithm 11.4 Nondeterministic knapsack algorithm

The for loop in line 4 is used to check if i^{th} item has to be placed in the Knapsack or not. Choice function in line 6 some how correctly determines if i^{th} item has to be placed in Knapsack or not and accordinly the weight W and profit P are updated. In line 9 we check if the total weight of all items selected into knapsack W doesnot exceed the capacity of the knapsack 'm' and the total profit earned is atleast 'r', if not the algorithm terminates unsuccessfully otherwise the algorithm terminates successfully. The time complexity of the above nondeterministic algorithm is O(n).

Satisfiability Problem (SAT):

Example 11.9 [Satisfiability] Let $x_1, x_2, \ldots$ denote boolean variables (their value is either true or false). Let $\bar{x}_i$ denote the negation of x_i. A *literal* is either a variable or its negation. A formula in the propositional calculus is an expression that can be constructed using literals and the operations **and** and **or**. Examples of such formulas are $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ and $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$. The symbol $\vee$ denotes **or** and $\wedge$ denotes **and**. A formula is in *conjunctive normal form* (CNF) if and only if it is represented as $\wedge_{i=1}^{k} c_i$, where the c_i are clauses each represented as $\vee l_{ij}$. The l_{ij} are literals. It is in *disjunctive normal form* (DNF) if and only if it is represented as $\vee_{i=1}^{k} c_i$ and each clause c_i is represented as $\wedge l_{ij}$. Thus $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x}_4)$ is in DNF whereas $(x_3 \vee \bar{x}_4) \wedge (x_1 \vee \bar{x}_2)$ is in CNF. The **satisfiability** problem is to determine whether a formula is true for some assignment of truth values to the variables. *CNF-satisfiability* is the satisfiability problem for CNF formulas.

It is easy to obtain a polynomial time nondeterministic algorithm that terminates successfully if and only if a given propositional formula $E(x_1, \ldots, x_n)$ is satisfiable. Such an algorithm could proceed by simply choosing (nondeter-

ministically) one of the 2^n possible assignments of truth values to $(x_1, \ldots, x_n)$ and verifying that $E(x_1, \ldots, x_n)$ is true for that assignment.

Eval (Algorithm 11.6) does this. The nondeterministic time required by the algorithm is $O(n)$ to choose the value of $(x_1, \ldots, x_n)$ plus the time needed to deterministically evaluate E for that assignment. This time is proportional to the length of E. □

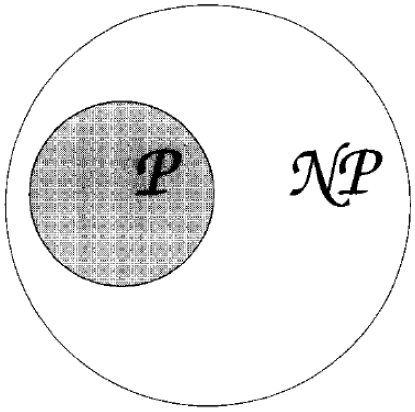
```
1   Algorithm Eval(E, n)
2   // Determine whether the propositional formula E is
3   // satisfiable. The variables are x1, x2, ..., xn.
4   {
5       for i := 1 to n do  // Choose a truth value assignment.
6           xi := Choice(false, true);
7       if E(x1, ..., xn) then Success();
8       else Failure();
9   }
```

Algorithm 11.6 Nondeterministic satisfiability

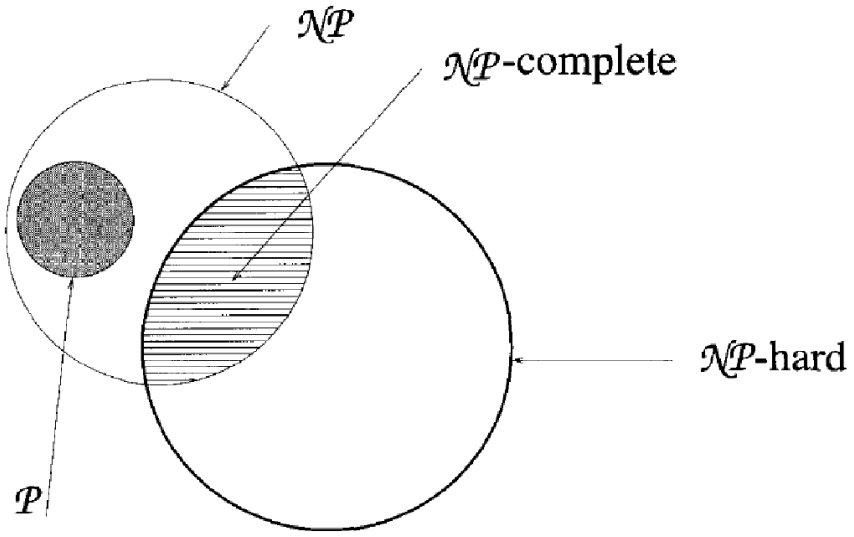
The deterministic algorithm for satisfiability takes $O(2^n)$ time because it has to check all possible values of given 'n' Boolean variables. In the non deterministic algorithm described above lines 5 nd 6 assigns either 0 or 1 (false or true) to each of the variables $x_1, x_2, x_3, x_{4\ldots}$. The above nondtermininstic algorithm time depends on line 7 which is used to check if the values assigned to each variable (either 0 or 1) makes the formula true or not.

Definition 11.3 $\mathcal{P}$ is the set of all decision problems solvable by deterministic algorithms in polynomial time. $\mathcal{NP}$ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time. □

Since deterministic algorithms are just a special case of nondeterministic ones, we conclude that $\mathcal{P} \subseteq \mathcal{NP}$. What we do not know, and what has become perhaps the most famous unsolved problem in computer science, is whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.



Definition 11.5 A problem L is $\mathcal{NP}$-hard if and only if satisfiability reduces to L (satisfiability $\propto L$). A problem L is $\mathcal{NP}$-complete if and only if L is $\mathcal{NP}$-hard and $L \in \mathcal{NP}$. □



If satisfiability (SAT) problem can be solved in polynomial time all exponential time algorithms can be solved in polynomial time.

SAT problem: Boolean satisfiability

Input is a Boolean formula. It has variables like x_1,x_2,x_3 …. If a Boolean formula has only 3 unique variables it is called 3-SAT. In Boolean formula x_1,x_2,x_3… can take 0 or 1.

The allowed operations are not $\overline{x_1}$, $\overline{x_2}$ …

And – $x_1 \wedge x_2$

Or – $x_1 \vee x_2$

So, the satisfiability problem is for what values of X_1, X_2, X_3 a formula like the following is true

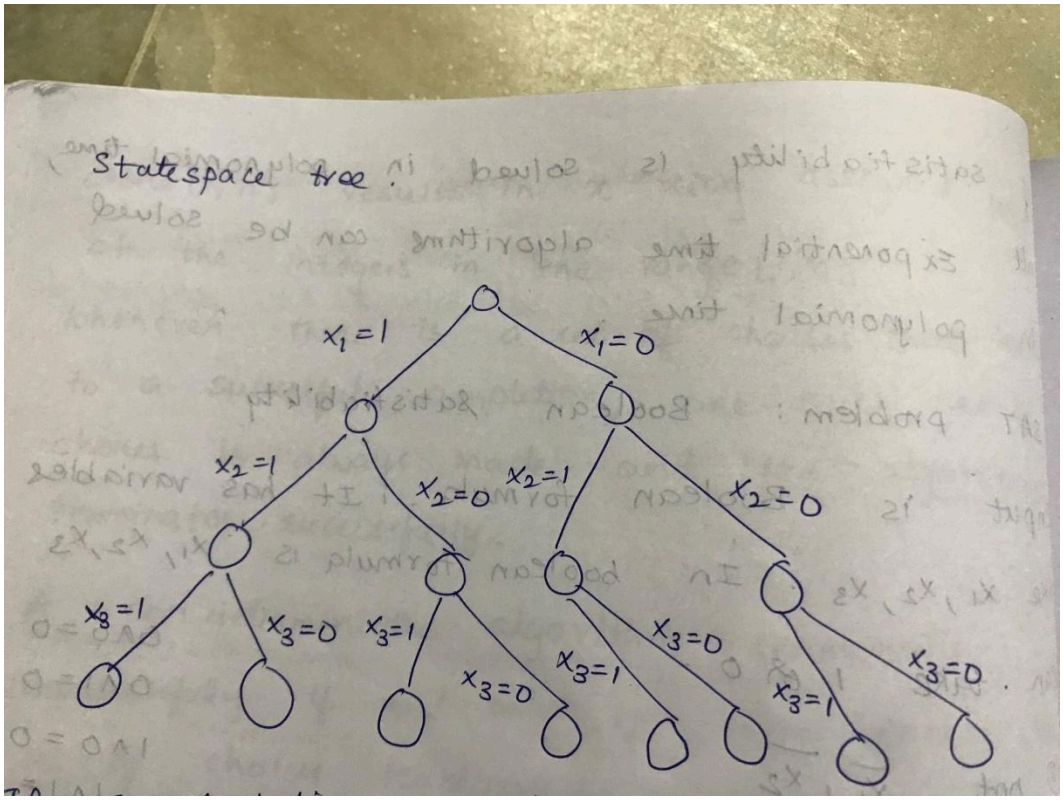
$$\left(\left(X_1 \vee X_2\right) \vee \overline{X_3}\right) \wedge \left(\overline{X_1} \vee X_2\right)$$

X_1	X_2	X_3
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Since there are 3 variables there are 2^3 possible combinations of values

If there are n variables we need to check 2^n possible combinations of values.

So, finding the answer takes exponential time.



The above diagram shows the state space tree for 3-SAT problem. If satisfiability can be solved in polynomial time all others can also be solved in polynomial time. If any one of them can be solved in polynomial all can be solved in polynomial time.

0/1 Knapsack problem

Let n=3, m=8, p={10,8,12}, w={5,4,3}

Solution can be expressed as $\{x_1, x_2, x_3\}$ i.e. which can be one of the following as solution set i.e. either {0,0,0} or {0,0,1} or {0,1,0} or {0,1,1}… ----------(1)

If there are n objects there 2^n combinations. This can be solved using state space tree using techniques like branch and bound or backtracking. ---------- (2)

Prove that 0/1 Knapsack is NP-Hard

Since Satisfiability is NP-hard and if Satifiability reduces (symbol $\propto$) to 0/1 Knapsack then 0/1 Knapsack will be NP-Hard.

Take an instance of satisfiabilty problem and convert it to 0/1 Knapsack problem. Conversion takes polynomial time. If 0/1 Knapsack problem can be solved in polynomial time, then satisfiability problem can also be solved in polynomial time. From (1) satisfiability can be reduced to 0/1 Knapsack . So, 0/1 Knapsack is also NP-hard.

Satisfiability is not only NP-hard but also has a non-deterministic polynomial time algorithm so it is NP-complete.

Cooks theorem:

[Cook] Satisfiability is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.

To find how satisfiability reduces to clique decision problem go through Abdul Bari Youtube channel.