UNDECIDABILITY

- **Recursive Language**
  - A language L is recursive, if L is the set of strings accepted by some Turing Machine (TM) that halts on every input.
  - The Turing Machine will halt every time and give answer(accepted or rejected) for each and every input string.
- **Recursive Enumerable Language**
  - A language L is recursively enumerable if L is the set of strings accepted by some TM.
  - But may or may not halt for all input strings which are not in ‘L’.

Decidable Language

- A language L is decidable if it is a recursive language. All decidable languages are recursive languages and vice-versa.
- A problem is said to be a Decidable problem if there exists a corresponding Turing machine which **halts** on every input with an answer- **yes or no**.
- It is also important to know that these problems are termed as **Turing Decidable** since a Turing machine always halts on every input, accepting or rejecting it.

Decidable Language

Example :

- Are two regular languages L and M equivalent?

  - We can easily check this by using Set Difference operation.

    L-M =Null and M-L =Null.

    Hence (L-M) U (M-L) = Null, then L,M are equivalent.

Semi - Decidable Language

- A languages L is semi decidable if L is a recursive enumerable language
- Semi-Decidable problems are those for which a Turing machine halts on the input accepted by it but it can either halt or loop forever on the input which is rejected by the Turing Machine.
- Such problems are termed as Turing Recognizable problems.

Undecidable Language

- A languages L is undecidable if it is not decidable
- An undecidable languages may sometimes be partially decidable but not decidable
- If a languages in not even partially decidable, then there exist no Turing machine for that language

Undecidable Language

Example:

- Whether a CFG generates all the strings or not?
  As a CFG generates infinite strings, we can’t ever reach up to the last string and hence it is Undecidable.
- Whether two CFG L and M equal?
  Since we cannot determine all the strings of any CFG, we can predict that two CFG are equal or not.

Church Thesis

- “No computational procedure will be considered an algorithm unless it can be represented as a Turing machine”
- This statement is called Church’s thesis because Alonzo Church(1936), gave many sophisticated reasons for believing it.
- Church’s original statement was slightly different because his thesis was presented slightly before the Alan turing invented his machines.

Church-Turing Thesis

- “A computation on numbers by an algorithm if and only if it is computable by Turing Machine”
- This statement has been given in the year 1936 by Alan Turing, but the fact is that till that time even the logic gate was also not available

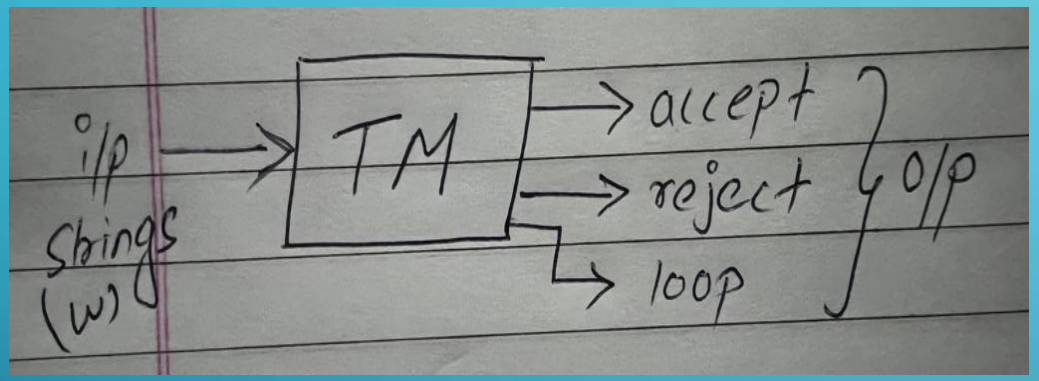
➢ Anything that can be done by current digital computer can also be done by a Turing machine

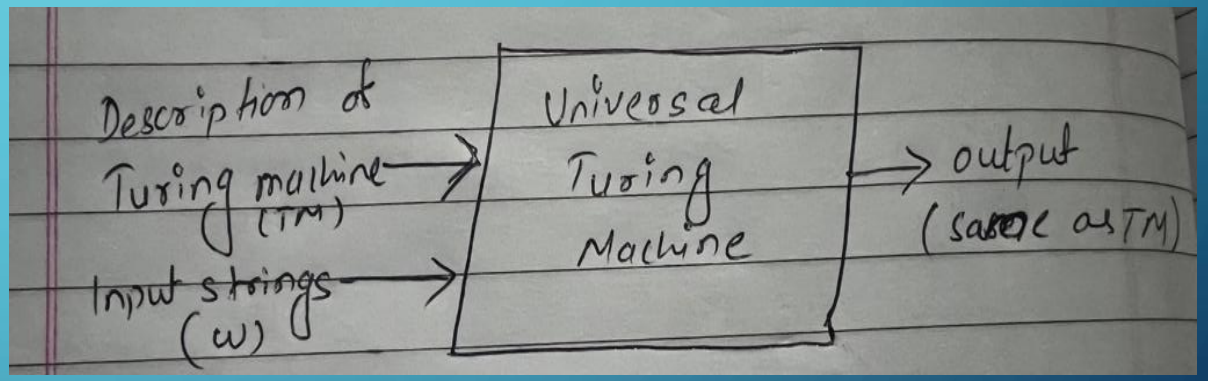
➢ Currently there is no problem which can be solved by a digital computer and can not be solved by a Turing Machine

➢ Many mathematical models are suggested but no one of them is more powerful than the Turing machine

Universal Turing Machine

- A Turing Machine is said to be Universal Turing Machine (UTM) if it can accept the input data and an algorithm(description) for computing





- TM is hardwired machine where UTM is programmable Turing machine
- TM does single task where UTM can do different type of task

Universal Turing Machine

- A Universal Turing Machine is a specified Turing Machine that can simulate the behavior of any Turing Machine
- To design a UTM, we have to increase the number of read/write heads, dimension of input tape & adding a special purpose memory in basic model of Turing Machine
- The problem with Turing Machine is that a different machine must be constructed for every new computation to be performed, that is for every input output relation
- To solve this problem of Turing Machine, Universal Turing machine introduced, which takes description of a machine M & input on the tape

Universal Turing Machine

- We design UTM then simulate TM on the content of input tape
  = Therefore UTM can simulate any other Turing Machine
- So for any problem that can be solved by Turing Machine, you can either use a Turing Machine that directly solves the problem or you could use a UTM 7 give it the description of TM that directly solves it.

Encoding functions :-

→ first associate a string of 0's to each state, each tape symbol and each of three directions.

→ Let, function S is defined as;

$S(B) = 0$

$S(a_i) = 0^{i+1}$ for each $a_i \in S$

$S(q_i) = 0^{i+2}$ for each $q_i \in Q$

$S(S) = 0$

$S(L) = 00$

$S(R) = 000$

Consider an example where TM T is defined as

$T = (\{q_1, q_2, q_3\}, \{a, b\}, \{a, b, B\}, \delta, q_1, B, F)$

where δ is defined as; $\delta(q_1, b) = (q_3, a, R)$

$\delta(q_3, a) = (q_1, b, R)$

$\delta(q_3, b) = (q_2, a, R)$

$\delta(q_3, B) = (q_1, b, L)$

Now, using encoding function as defined above.

$S(q_1) = 000$

$S(q_2) = 0000$

$S(q_3) = 00000$

$S(a_1) = 00$: say $a_1 = a$

$S(a_2) = 000$ $a_2 = b$

$S(B) = 0$

$S(R) = 000$

$S(L) = 00$

$S(S) = 0$

Now, for m_1

$e(m_1) = s(q_1)1\,s(b)1\,s(q_1)1\,s(a)1\,s(R)$

$= 0001000100000 1001000$

for m_2

$e(m_2) = s(q_2)1\,s(a)1\,s(q_1)1\,s(b)1\,s(R)$

$= 000001001000 10001000$

for m_3

$e(m_3) = s(q_3)1\,s(b)1\,s(q_2)1\,s(a)1\,s(R)$

$= 0000010001000 01000$

for m_4

$e(m_4) = s(q_3)1\,s(B)1\,s(q_3)1\,s(b)1\,s(L)$

$= 000001 01 000001000100$

Now,

Code for TM T

$e(T) = e(m_1)11e(m_2)11e(m_3)11\,e(m_4)$

$= 0001000100060100100011.$

000001001000100010011

$0000010001000010001 1$

$0000010100000 1000100$

Now,

for T and any i/p string x where $x = ab$

code will be $= e(T)111e(x)$; where $e(x) = s(a)1\,s(b)$

$= 001000$

$= 0001000100000100100011000001001000 1$

$000100011000001000100001000110000010100000 1$

$000100 111 001000$

Halting problem of Turing machine

- The halting problem of turing machine is that “ For a given program/algorithm, it is not possible to determine that whether the turing machine will ever halt or not ”
- Halting means that program on certain input will accept it and halt, or reject it and halt. It will never go into infinite loop
- Halting actually means terminating.
- So can we have an algorithm that will tell that the given program will halt or not. In terms of turing machine, will it terminate when run on a turing machine with some particular given input string

  - The answer is no, i.e. we can not design a generalized algorithm which can appropriately say that given a program will ever halt or not

Halting problem of Turing machine

- So this is an Undecidable problem because we can not have an algorithm which tell us whether a given Turing machine will halt or not in a generalized way
- The best possible solution is to run a program or turing machine and see whether it halts or not.

Recursive functions theory

Initial Functions

- There are three initial functions

1. Zero functions(Z)
   - It is defined as Z(x) = 0
   - For eg: Z(5) = 0
2. Successor function(S)
   - It is defined as S(x) = x + 1
   - For eg: S(6) = 6 + 1 = 7
3. Projection functions (P)
   - It is defined as $P_i^n(x_1, x_2, x_3, x_4 \ldots . x_n) = x_i$
   - For eg: $P_2^4(1,3,5,7) = 7$

Recursive functions theory

Composite functions

- A composite function is defined when one functions is substituted into another function.
- For eg: Let f(x) = 3x+1

  g(x) = x+5

  then, f(g(x)) = f(x+5) = 3(x+5)+1 = 3x+16

  or g(f(x)) = g(3x+1) = (3x+1)+5 = 3x+6

Recursive functions theory

Recursive function

- A function that repeats itself again and again is called recursive functions
- Types:
  1. Primitive Recursive function
  2. Partial Recursive function
  3. Total Recursive function

Recursive functions theory

1. Primitive Recursive function

- A function is called primitive recursive function, if it can be obtained from initial functions through finite number of composition and recursive steps.
- For eg: Addition of integer (x,y)

  add(x,0) = x …….. (1)

  add(x,y+1) = add(x,y) + 1 ……(2)

  for add 3 and 2

  add(3,2) = add(3,1+1)

  = add(3,1) + 1 = add(3,0+1) + 1

  = add(3,0) + 1 + 1

  = 3 + 1 + 1

  = 5

Recursive functions theory

1. Primitive Recursive function

- For eg: Multiplication of integer (x,y)

  mult(x,0) = 0 …….. (1)

  mult(x,y+1) = add(x,mult(x,y)) ……(2)

  for add 3 and 2

  mult(3,2) = mult(3,1+1) =add(3,mult(3,1)) = add(3,mult(3,0+1))

  = add(3,add(3,mult(3,0)))

  = add(3,add(3,0))

  = add(3,3)

  = add(3,2) + 1

  = add(3,1)+1+1 = add(3,0)+1+1+1 = 3+1+1+1 = 6

Recursive functions theory

2. Partial Recursive function

- A functions is called partial recursive function if it is defined only for some of its arguments.

  For eg: subtraction of two positive integer number m and n

  f(m,n) = m – n, m >= n

  m < n

- For eg: multiplication of two positive integer number m and n
- f(m,n) = m * n, m >= n

  m < n

Recursive functions theory

3. Total Recursive function

- A recursive function is said to be total recursive function, if it is designed for all its arguments.
- So a total function is said to be primitive recursive function if and only if it is an initial functions obtained by applying composition and recursion with finite no. steps
- For eg: multiplication of two positive no’s is total recursive and primitive recursive

COMPUTATIONAL COMPLEXITY

COMPUTATIONAL COMPLEXITY

- The complexity theory provides the theoretical estimates for the resources needed by an algorithm to solve any computational task
- It involves classifying problem according to their tractability or intractability i.e. whether they are easy or hard to solve
- The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n).
- The complexity of an algorithm can be divided into two types.
  - The time complexity and the space complexity.

COMPLEXITIES OF AN ALGORITHM

Time Complexity

- The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm.
- This calculation is totally independent of implementation and programming language.

Space Complexity

- Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm.
- The memory space is generally considered as the primary memory.

ALGORITHM

- An algorithm is a finite set of instructions, those if followed, accomplishes a particular task.
- It is not language specific, we can use any language and symbols to represent instructions.

Algorithm

Polynomial time	Exponential Time
n - Linear Search logn - Binary Search n*n - Insertion Sort n*logn - Merge Sort n*n*n - Matrix Multiplication	2^n - 0/1 knapsack 2^n - Travelling SP 2^n - Sum of Subsets 2^n - Graph Coloring 2^n - Hamilton Cycle

TRACTABILITY AND INTRACTABILITY

- A problem is called tractable iif there is an efficient algorithm that solves it in polynomial time
- Eg: Linear search (O(n)), binary search (O(logn)), merge sort(O(nlogn))

- There is no efficient algorithm to solve it or A problem that cant be solved in polynomial time but verifiable in polynomial time
- Eg: Travelling sales man, Graph coloring, scheduling, SU-DU-KU = (2^n)

TRACTABILITY AND INTRACTABILITY

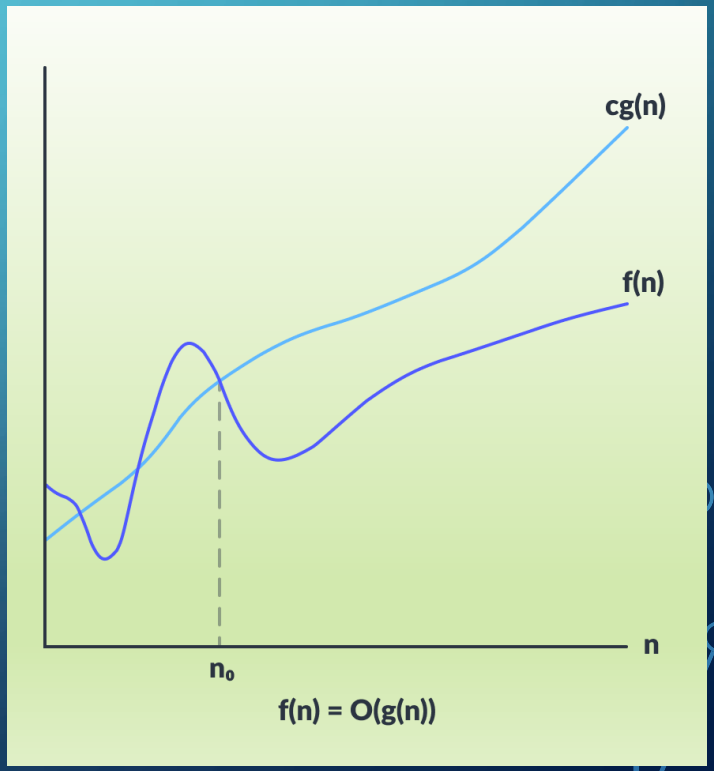
- EG: SU-DU-KU

		4		5				
9			7	3	4	6		
		3		2	1		4	9
	3	5		9		4	8	
	9						3	
	7	6		1		9	2	
3	1		9	7		2		
		9	1	8	2			3
				6		1		

BIG OH 'O' NOTATION

- Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

- $O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$
  - The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.
  - For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).
  - Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.



CLASS P AND NP PROBLEM

P Class – Problem

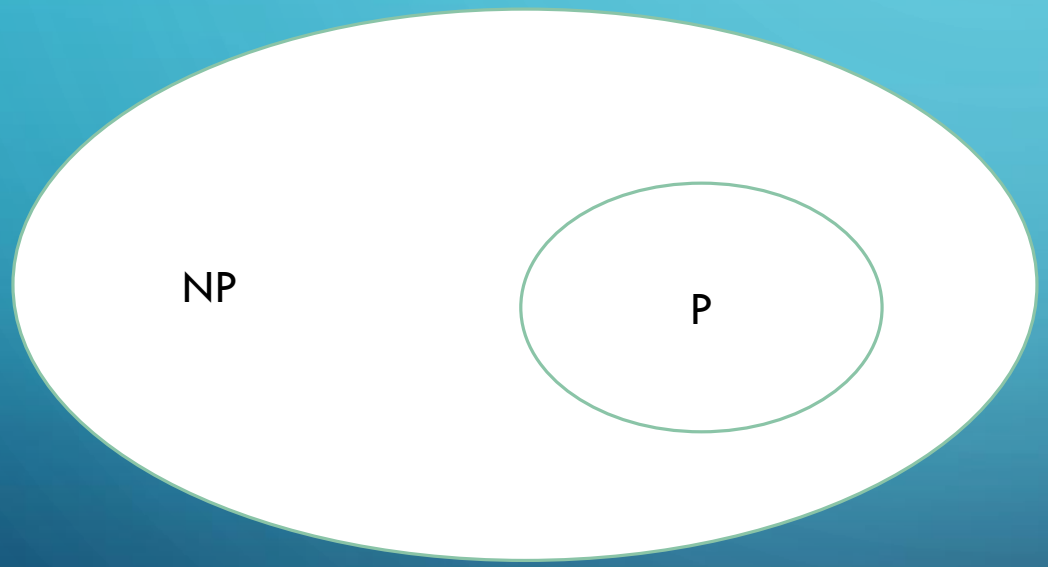
- P is set of problems that can be solved(deterministic) in polynomial [P] time
- Eg: Linear search (O(n)), binary search (O(logn)), merge sort(O(nlogn))
- Formally, an algorithm is polynomial time algorithm, if there exists a polynomial p(n) such that the algorithm can solve any instance of size n in a time O(p(n))

CLASS P AND NP PROBLEM

NP Class – Problem

- NP is set of problems that can be solved (non-deterministic) in polynomial time
- Also NP problem can be verified in polynomial time
- Eg: Travelling sales man, 0/1 knapsack, graph coloring, Hamilton cycle, SUDUKU
-

CLASS P AND NP PROBLEM

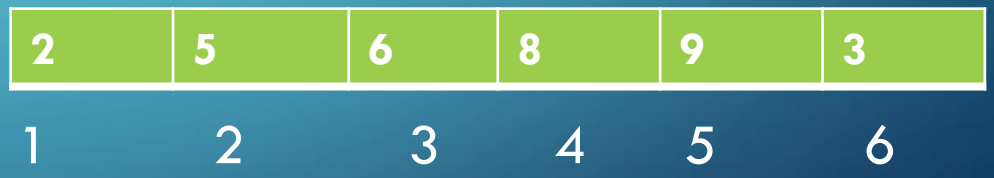


- P class problems are subset of NP class problem
- Does P=NP?
- P!=np

NON DETERMINISTIC ALGORITHM

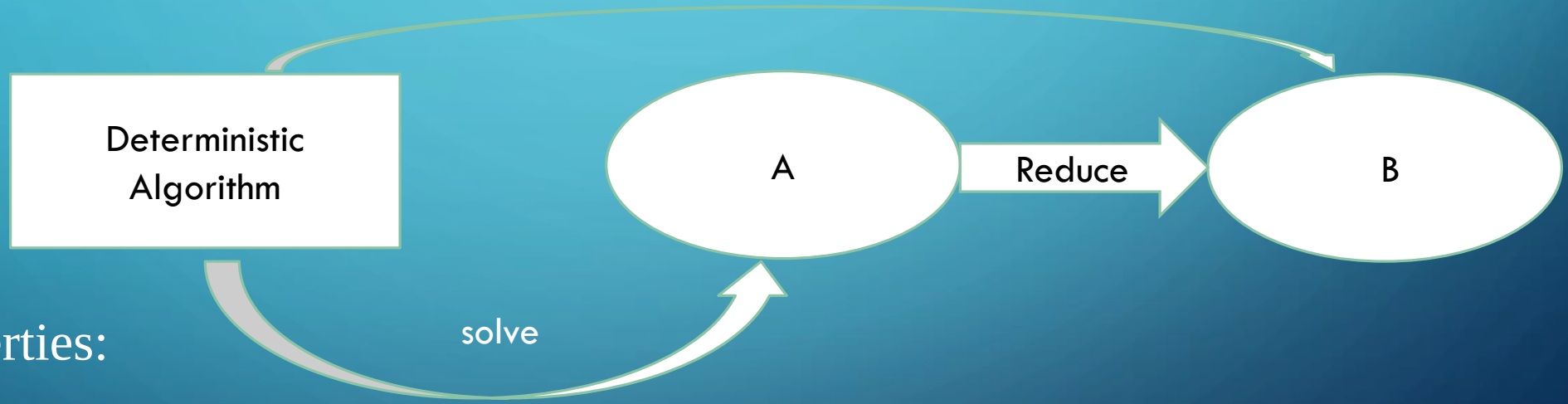
```
Baral Search (A, n, Key)          key = 9       1
 {
   i = choice();      5     -------O(1)
   if(A[i]==key)  6==6     ---O(1)
     {
       print(i)  //success
     }
    print(0)    //failure
  }
```

A Key = 6



REDUCTION

- Let A and B are two problems in NP. If problem A is reduce to problem B, iff there is a way to solve A by deterministic algorithm that solve B in polynomial time. The we can denote A ∝ B

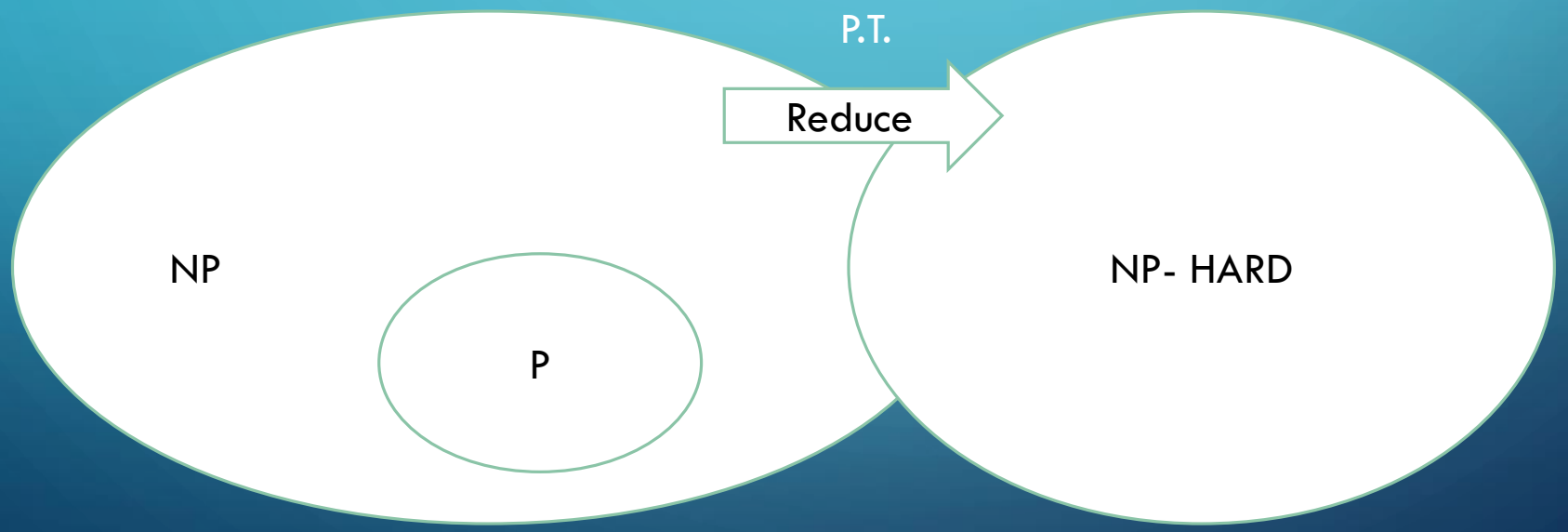


- Properties:
  - If A is reducible to B and B is in polynomial time, then A also in polynomial time
  - A is not in polynomial time, It implies that B is not in polynomial time

NP-HARD AND NP-COMPLETE

NP-Hard Problem

- Every problem in NP class can be reduced into other set using polynomial time, then it is called as NP- Hard problem



NP-HARD AND NP-COMPLETE

NP-Complete

- The group of problems which are both in NP and NP-hard are known as NP complete problem
- The NP-Complete problems are NP-Hard but not all NP-Hard problems are not NP-Complete
- If you write a non deterministic polynomial time algorithm for np hard problem is called np complete= o/1 knapsack problem reduce (PT) in Hamilton cycle = Non deterministic manner using polynomial time