



# Data Structures and Algorithms Design

**BITS Pilani**  
Hyderabad Campus

Febin.A.Vahab

# CONTACT SESSION 14 -PLAN



Contact Sessions(#)	List of Topic Title	Text/Ref Book/external resource
14	<b>Definition of P and NP classes and examples, Understanding NP-Completeness: CNF-SAT Cook-Levin theorem</b> <b>Polynomial time reducibility: CNF-SAT and 3-SAT, Vertex Cover</b>	<b>T1: 13.1, 13.2, 13.3</b>

# P and NP classes



0 0 0 1  
0 0 1 0  
1 1 1 1  
 $2^m$

nW

Polynomial Time	Exponential Time
Linear Search $O(n)$	0/1 Knapsack $O(2^n)$
Binary Search $O(\log n)$	Tower of Hanoi $O(2^n)$
Merge Sort $O(n \log n)$	TSP $O(2^n)$
Quicksort $O(n \log n), O(n^2)$	Sum of Subsets $O(2^n)$
Fractional KS $O(n \log n)$	
Task Scheduling $O(n^2)$	
Kruskal's MST $O(m \log n)$	

TSP

# Decision and Optimization Problems



- **Decision Problem:** computational problem with intended output of “yes” or “no”, 1 or 0
- **Optimization Problem:** computational problem where we try to maximize or minimize some value.
- Introduce parameter  $k$  and ask if the optimal value for the problem is at most or at least  $k$ . Turn optimization into decision

# Decision and Optimization Problems



- Example: Hamiltonian circles:
- A Hamiltonian circle in an undirected graph is a simple circle that passes through every vertex exactly once.
- The **decision** problem is: Does a given undirected graph has a Hamiltonian circle?

# Decision and Optimization Problems



- Many problems will have decision and optimization versions
- **Traveling salesman problem**
- Optimization problem: Given a weighted graph, find Hamiltonian cycle of minimum weight.
- Decision problem: Given a weighted graph and an integer  $k$ , is there a Hamiltonian cycle with total weight at most  $k$ ?

# Decision and Optimization Problems



- **Knapsack** : Suppose we have a knapsack of capacity  $W$  and  $n$  objects with weights  $w_1, \dots, w_n$ , and values  $v_1, \dots, v_n$
- **Optimization problem**: Find the largest total value of any subset of the objects that fits in the knapsack (and find a subset that achieves the maximum value)
- **Decision problem**: Given  $k$ , is there a subset of the objects that fits in the knapsack and has total value at least  $k$ ?

# Decision and Optimization Problems



- **Graph coloring:** assign a color to each vertex so that adjacent vertices are not assigned the same color. Chromatic number: the smallest number of colors needed to color  $G$ .
- We are given an undirected graph  $G = (V, E)$  to be colored.
- **Optimization problem:** Given  $G$ , determine the chromatic number .
- **Decision problem:** Given  $G$  and a positive integer  $k$ , is there a coloring of  $G$  using at most  $k$  colors? If so,  $G$  is said to be  $k$ -colorable



# Decision and Optimization Problems



- **Decision Problem:** computational problem with intended output of “yes” or “no”, 1 or 0

Examples:

- Does a text  $T$  contain a pattern  $P$ ?
- Does an instance of 0/1 Knapsack have a solution with benefit at least  $K$ ?
- Does a graph  $G$  have an MST with weight at most  $K$ ?

# Polynomial-Time



- The class P consists of those problems that are solvable in polynomial time.
- More specifically, they are problems that can be solved in time  $O(n^k)$  for some constant  $k$ , where  $n$  is the size of the input to the problem
- The key is that  $n$  is the **size of input**

# Problems and Languages



- A **language**  $L$  is a set of strings defined over some alphabet  $\Sigma$
- Every decision algorithm  $A$  defines a language  $L$ 
  - $L$  is the set consisting of every string  $x$  such that  $A$  outputs “yes” on input  $x$ .
  - We say “ $A$  **accepts**  $x$ ” in this case
    - Example:
    - If  $A$  determines whether or not a given graph  $G$  has an Euler tour, then the language  $L$  for  $A$  is all graphs with Euler tours.

# The Complexity Class P

- A **complexity class** is a collection of languages
- P is the complexity class consisting of all languages that are accepted by **polynomial-time** algorithms
- For each language L in P there is a polynomial-time decision algorithm A for L.
  - If  $n=|x|$ , for  $x$  in L, then A runs in  $p(n)$  time on input  $x$ .
  - The function  $p(n)$  is some polynomial

# The Complexity Class NP

---

- **NP is not the same as non-polynomial complexity/running time.**
- **NP does not stand for not polynomial.**
- **NP = Non-Deterministic polynomial time**

# Deterministic



- In a deterministic computer we can determine in advance for every computational cycle, the output state by looking at the input state
- In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states

# Deterministic



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

//output: the position of  $el$  in  $A$

*search( $el, A, i$ )*

{

*if ( $A[i] = el$ ) then return  $i$*

*else*

*return search( $el, A, i+1$ )*

}

Complexity:  $O(N)$

$el = 9$

↓ ↓ ↓  
7 5 3 8 9 3

# Non Deterministic



- In a nondeterministic computer we may have more than one output state.
- The computer “chooses” the correct one (“nondeterministic choice”)
- For the same input, the compiler may produce different output in different runs.
- Non-deterministic algorithms can’t solve the problem in polynomial time and can’t determine what is the next step



# Non Deterministic



- Non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.
- A **nondeterministic algorithm** for a problem **A** is a two-stage procedure. **In the first phase**, a procedure makes a guess about the possible solution for A. **In the second phase**, a procedure checks if the guessed solution is indeed a solution for A

# Non Deterministic



- **Search an element  $x$  on  $A[1:n]$  where  $n \geq 1$ , on successful search return  $j$  if  $a[j]$  is equals to  $x$  otherwise return  $0$ .**

*$j = \text{choice}(a, n)$*

*if ( $A[j] == x$ ) then*

*{*

*write( $j$ );*

*success();*

*}*

*write( $0$ ); failure();*

Complexity:  $O(1)$

# Non Deterministic



- **choice(X)** : chooses any value randomly from the set X.
- **failure()** : denotes the unsuccessful solution.
- **success()** : Solution is successful and current thread terminates.

## Non Deterministic Sorting

---

```
1. Algorithm Nsort(A,N)
2. // sort n positive numbers
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 the order
12.  If B[i] > B[i+1] then failure ();
13.  Write (B);
14.  Success();
15. }
```

- In the for loop of 5 to 10 each  $A[i]$  is assigned to position in B
- Line 7 non deterministically identifies this position
- Line 8 checks whether this position is already used or not !
- The order of numbers in B is some permutation of the initial order in A
- For loop of line no 11 and 12 verifies the B is in ascending order,
- Since there is always a set of choices at line no 7 for ascending order, this is a sorting algorithm of Complexity  $O(N)$
- All deterministic sorting algorithms must have A complexity  $O(n \log n)$

# The Complexity Class NP

- The class **NP** consists of all problems that can be solved in polynomial time by **nondeterministic algorithms**
- That is, both phase 1 and phase 2 run in polynomial time
- If  $A$  is a problem in  $P$  then  $A$  is a problem in  $NP$  because
  - Phase 1: use the polynomial algorithm that solves  $A$
  - Phase 2: write a constant time procedure that always returns true

# The Complexity Class NP

- Every decision problem solvable by a polynomial time deterministic algorithm is also solvable by a polynomial time nondeterministic algorithm.
- To see this, observe that any deterministic algorithm can be used as the checking stage of a nondeterministic algorithm.
- If  $I \in P$ , and  $A$  is any polynomial deterministic algorithm for  $I$ , we can obtain a polynomial nondeterministic algorithm for  $I$  merely by using  $A$  as the checking stage and ignoring the guess.
- Thus  $I \in P$  implies  $I \in NP$

# The Complexity Class NP

---

- The key question is are there problems in NP that are not in P or is  $P = NP$ ?
- We don't know the answer to the previous question
- We say that a non-deterministic algorithm A **accepts** a string  $x$  if there exists some sequence of choose operations that causes A to output “yes” on input  $x$ .
- NP is the complexity class consisting of all languages accepted by **polynomial-time non-deterministic** algorithms.

# The Complexity Class NP



- How to prove that a problem  $A$  is in NP:

Show that  $A$  is in P

OR

Write a nondeterministic algorithm solving  $A$  that runs in polynomial time



# NP example



Problem: Decide if a graph has an MST of weight  $K$

Algorithm:

1. Non-deterministically choose a set  $T$  of  $n-1$  edges
2. Test that  $T$  forms a spanning tree
3. Test that  $T$  has weight at most  $K$

Analysis: Testing takes  $O(n+m)$  time, so this algorithm runs in polynomial time.

# The Complexity Class NP Alternate Definition



- We say that an algorithm B **verifies** the acceptance of a language L if and only if, for any x in L, there exists a certificate y such that B outputs “yes” on input (x,y).
- NP is the complexity class consisting of all languages verified by **polynomial-time** algorithms.
- We know: P is a subset of NP.
- Major open question:  $P=NP$ ?
- Most researchers believe that P and NP are different

# Satisfiability Problem

- A Boolean formula is *satisfiable* if there exists some assignment of the values 0 and 1 to its variables that causes it to evaluate to 1.
- CNF – Conjunctive Normal Form. ANDing of clauses of ORs

$$(x_0 \vee x_2) \wedge (\neg x_0 \vee x_1) \wedge (x_0 \vee x_1 \vee \neg x_2)$$

# CNF satisfiability



- This problem is in *NP*. Nondeterministic algorithm:
  - Guess truth assignment
  - Check assignment to see if it satisfies CNF formula
  - It is easy to show that CNF-SAT is in NP
  - For, given a Boolean formula  $S$ , we can construct a simple nondeterministic algorithm that first "guesses" an assignment of Boolean values for the variables in  $S$  and then evaluates each clause of  $S$  in turn.
  - If all the clauses of  $S$  evaluate to 1, then  $S$  is satisfied; otherwise, it is not.

# CNF satisfiability



- Example:  
 $(A \vee \neg B \vee \neg C) \wedge (\neg A \vee B) \wedge (\neg B \vee D \vee F) \wedge (F \vee \neg D)$
- Truth assignments:

	A	B	C	D	E	F
--	---	---	---	---	---	---

1.	0	1	1	0	1	0
----	---	---	---	---	---	---

2.	1	0	0	0	0	1
----	---	---	---	---	---	---

3.	1	1	0	0	0	1
----	---	---	---	---	---	---

4. ... (how many more?)

Checking phase:  $\Theta(n)$

# 3 SAT

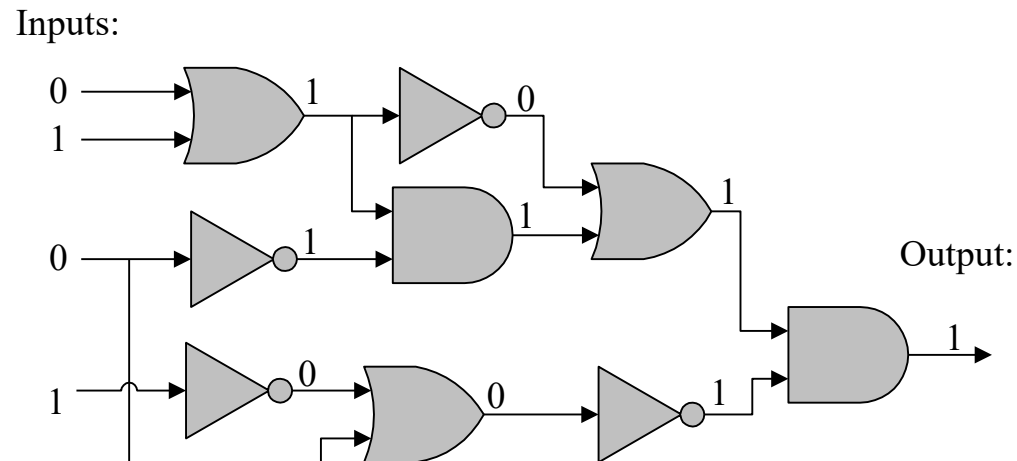
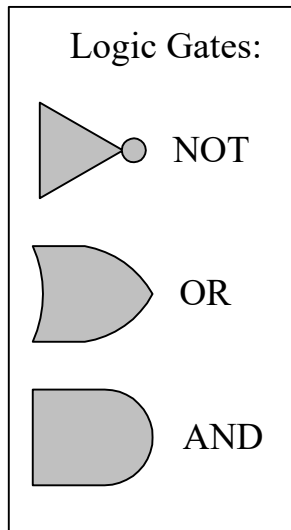


- Problem: Given a CNF where each clause has 3 variables, decide whether it is satisfiable or not.
- $(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$
- 3SAT is NP Complete

# An Interesting Problem



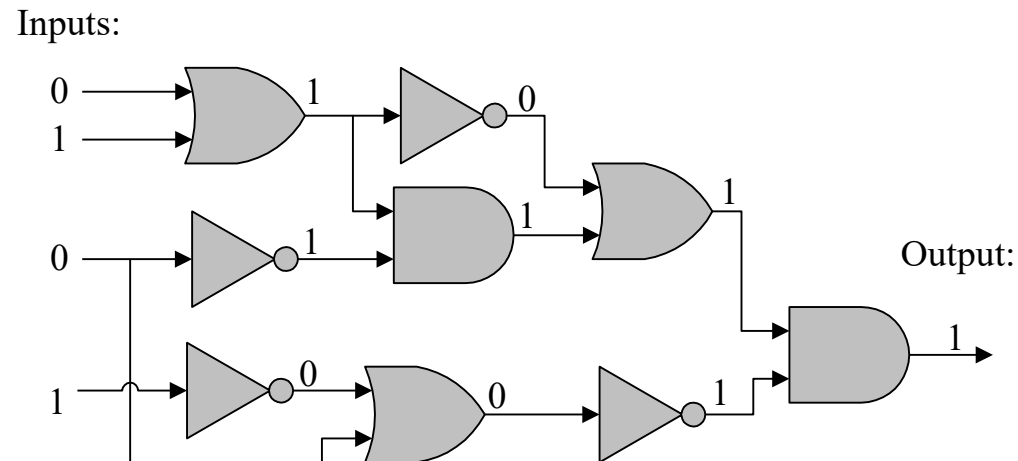
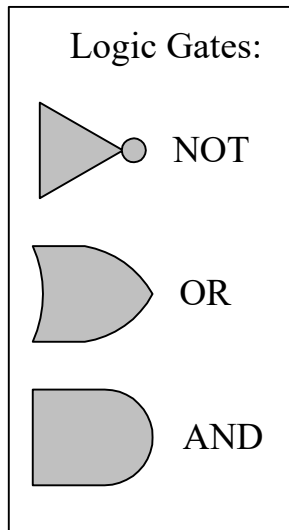
- A Boolean circuit is a circuit of AND, OR, and NOT gates; the CIRCUIT-SAT problem is to determine if there is an assignment of 0's and 1's to a circuit's inputs so that the circuit outputs 1.



# CIRCUIT-SAT is in NP



- ◆ Non-deterministically choose a set of inputs and the outcome of every gate, then test each gate's I/O.





# P And NP Summary



- **P** = set of problems that can be solved in polynomial time
  - Examples: Fractional Knapsack, ...
- **NP** = set of problems for which a solution can be verified in polynomial time
  - Examples: 0/1 Knapsack,..., Hamiltonian Cycle, CNF SAT, 3-CNF SAT
- Clearly **P**  $\subseteq$  **NP**
- Open question: Does **P** = **NP**?
  - Most suspect not
  - An August 2010 claim of proof that  $P \neq NP$ , by Vinay Deolalikar, researcher at HP Labs, Palo Alto, has flaws

# Polynomial-Time Reducibility



- Language  $L$  is polynomial-time reducible to language  $M$  if there is a function computable in polynomial time that takes an input  $x$  of  $L$  and transforms it to an input  $f(x)$  of  $M$ , such that  $x$  is a member of  $L$  if and only if  $f(x)$  is a member of  $M$ .
- Shorthand,  $L^{\text{poly}}M$  means  $L$  is polynomial-time reducible to  $M$

# Polynomial-Time Reducibility



- A problem R can be *reduced* to another problem Q if any instance of R can be rephrased to an instance of Q, the solution to which provides a solution to the instance of R
  - This rephrasing is called a *transformation*
- Intuitively: If R reduces in polynomial time to Q, R is “no harder to solve” than Q
- Example:  $\text{lcm}(m, n) = m * n / \text{gcd}(m, n)$ ,  
lcm(m,n) problem is reduced to gcd(m, n) problem

# Polynomial-Time Reducibility



- For example a Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.
- Hamiltonian circuit problem(HCP): Does a given undirected graph have a Hamiltonian cycle?
- Traveling salesman problem(TSP): Given a weighted graph and an integer  $k$ , is there a Hamiltonian cycle with total weight at most  $k$ ?

# NP Hard and NP complete

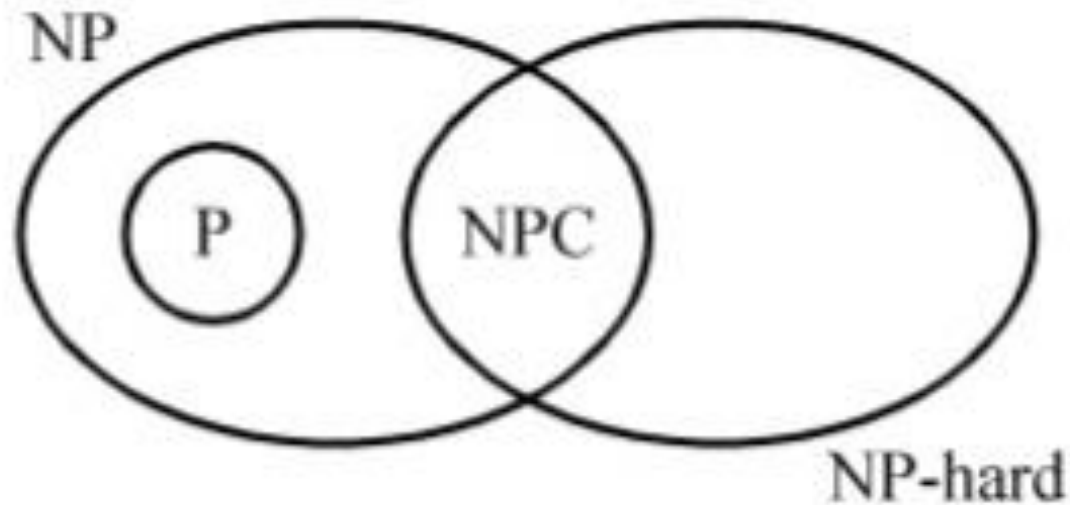


- A decision problem  $D$  is NP-complete iff
  1.  $D \in NP$
  2. every problem in  $NP$  is polynomial-time reducible to  $D$

# NP Hard and NP complete

- If  $R$  is *polynomial-time reducible* to  $Q$ , we denote this  $R \leq_p Q$
- Definition of **NP-Hard** and **NP-Complete**:
  - If all problems  $R \in \mathbf{NP}$  are *polynomial-time* reducible to  $Q$ , then  $Q$  is *NP-Hard*
  - We say  $Q$  is *NP-Complete* if  $Q$  is NP-Hard **and**  $Q \in \mathbf{NP}$
- If  $R \leq_p Q$  and  $R$  is NP-Hard,  $Q$  is also NP-Hard

# NP Hard and NP complete







# NP Hard and NP complete

---

- Belief:  $P$  is a proper subset of  $NP$ .
- Implication: the  $NP$ -complete problems are the hardest in  $NP$ .
- Why: Because if we could solve an  $NP$ -complete problem in polynomial time, we could solve every problem in  $NP$  in polynomial time.
- That is, if an  $NP$ -complete problem is solvable in polynomial time, then  $P=NP$ .
- Since so many people have attempted without success to find polynomial-time solutions to  $NP$ -complete problems, showing your problem is  $NP$ -complete is equivalent to showing that a lot of smart people have worked on your problem and found no polynomial-time algorithm

## Cook's theorem

NP = P iff the satisfiability problem is a P problem.

- SAT is NP-complete.
- It is the first NP-complete problem.
- Every NP problem reduces to SAT.



**Stephen Arthur  
Cook**

# Amusing analogy

(thanks to lecture notes at University of Utah)



- Students believe that every problem assigned to them is NP-complete in difficulty level, as they have to find the solutions. 😊
- Teaching Assistants, on the other hand, find that their job is only as hard as NP, as they only have to verify the student's answers.
- When some students confound the TAs, even verification becomes hard