34.1-1

Define the optimization problem LONGEST-PATH-LENGTH as the relation that associates each instance of an undirected graph and two vertices with the number of edges in a longest simple path between the two vertices. Define the decision problem LONGEST-PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ is an undirected graph, $u, v \in V$, $k \geq 0$ is an integer, and there exists a simple path from u to v in G consisting of at least k edges$\}$. Show that the optimization problem LONGEST-PATH-LENGTH can be solved in polynomial time if and only if LONGEST-PATH $\in$ P.

Suppose that a polynomial time taken to solve the LONGEST-PATH-LENGTH problem:

Suppose L denotes an algorithm for LONGEST-PATH-LENGTH problem. Now consider the following algorithm $L'$to decide the LONGEST-PATH.

Algorithm:

Input: The inputs taken are $\langle G,u,v,k\rangle$

Output: Here, the output will be "No" or "Yes".

1. $L'\langle G,u,v,k\rangle$
2. $\max = L(G,u,v)$;
3. **if** $L(k \leq \max)$ **then**
4. **return** "Yes"
5. **else**
6. **return** "No"

Since, L denotes an algorithm for LONGEST-PATH-LENGTH problem, it takes a polynomial time to run. In the above algorithm, L'simply calls the Lthen the algorithm L'will also take a polynomial running time.

Suppose that a polynomial time taken to decide the LONGEST-PATH:

Suppose algorithm L'takes a polynomial time to decide the LONGEST-PATH. Now, consider the following algorithm $L$to solve the LONGEST-PATH-LENGTH problem.

Algorithm:

Input: The inputs taken are $\langle G,u,v\rangle$

Output: Here, the output will be the size of the longest path exists between the vertices u and v or return -1 if there exists no path between these vertices.

1. $L\langle G,u,v\rangle$
2. $k = n-1$;
3. **while** ($k \geq 0$) and $(L'(G,u,v,k))$=false
4. **do**
5. $k = k-1$;
6. **return** k;

Now consider the above algorithm. Here, the algorithm $L'$are called at most $n$times and also some polynomial number of steps. Since, the algorithm L' takes a polynomial running time, then the algorithm Lwill also be run in polynomial time.

Hence, from the above explanation, **a polynomial time can be used to solve the optimization problem LONGEST-PATH-LENGTH if and only if LONGEST-PATH** $\in$ **P.**

34.1-4

Is the dynamic-programming algorithm for the 0-1 knapsack problem that is asked for in Exercise 16.2-2 a polynomial-time algorithm? Explain your answer.

The dynamic programming algorithm of running time O(nw) is not a polynomial time algorithm, because W is the maximum weight of items that the thief can put into his knapsack without loss of generality, assume that the input of the problem w is given as a binary number then the size of the input is O (n log w) the running time O(nw) is not a polynomial in O (n log w) because for any fixed constant C, nw > $(n \log w)^c$ for sufficiency large w.

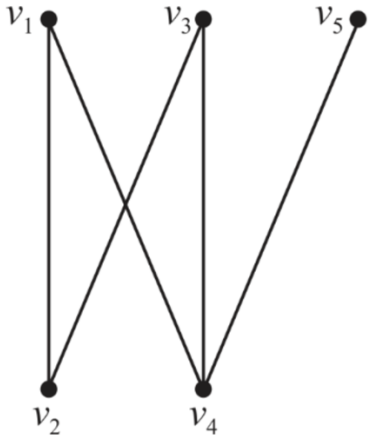
34.2-2

Prove that if G is an undirected bipartite graph with an odd number of vertices, then G is nonhamiltonian.

Hamiltonian graph: A graph that possesses a Hamiltonian cycle is called a Hamiltonian graph. In the absence of a Hamiltonian cycle, the graph is called non-Hamiltonian.

Hamiltonian cycle: A Hamiltonian cycle is a circuit or cycle that that visits each and every node of a graph exactly once.

• Bipartite graph is a graph in which the vertices can be divided into two disjoint sets such that an edge originating in one disjoint set always moves into another disjoint set.



The above diagram is an example of bipartite graph. The vertices are divided into two disjoin sets such that edge from one set always moves into vertex of another set.

• Owing to the above definition, every vertex has its adjacent vertex only in the disjoint set other than its own.

• So, in a Hamiltonian cycle, the edge must move from one disjoint set to another all the time.

• Eventually, the edge must come back to the original starting vertex. This requires equal number of vertices to be present in both disjoint sets so that the edge can travel from one disjoint set to other and alternate between the two and finally come back to the originating vertex.

So, $\text{No. of vertices in disjoint set 1} = \text{No. of vertices in disjoint set 2}$

$$\text{Even} + \text{Even} = \text{Even}$$

Since, $\text{Odd} + \text{Odd} = \text{Even}$, therefore, even number of vertices must present in a bipartite graph for it to be Hamiltonian. If number of vertices is Odd, then it can't be Hamiltonian.

Thus, it is proved that an undirected bipartite graph which has an odd number of vertices is always non-Hamiltonian.

34.2-3

Show that if HAM-CYCLE $\in$ P, then the problem of listing the vertices of a hamiltonian cycle, in order, is polynomial-time solvable.

Consider HAM–CYCLE = $\{\langle G\rangle : G \text{ is a hamiltonian graph}\}$ and HAM–CYCLE $\in P$.

For each node, exactly two incident edges participate in the cycle.

Define an algorithm CHECK–HAM–CYCLE(G') to find the Hamiltonian cycles.

• The algorithm takes any graph G'.

• It should return true if G' have a Hamiltonian cycle.

• It should return false if G' doesn't have a Hamiltonian cycle.

• It should be a polynomial–time algorithm.

Now design another algorithm GENERATE–HAM–CYCLE(G) that generates the order of a legitimate Hamiltonian cycle of G if exist.

Algorithm CHECK–HAM–CYCLE(G'):

1 select a node $v \in V$

2 E_v is the edges incident to the node v.

3 calculate the pair $e_1, e_2 \in E_v$ such that

$G' = (V, (E - E_v) \cup \{e_1, e_2\})$ contains a Hamiltonian cycle.

4 if Hamiltonian cycle exists then

5 return true

6 else

7 return false

Clearly, the calculation of Hamiltonian cycle with all possible pairs requires polynomial time. So the algorithm is a polynomial time algorithm.

Algorithm GENERATE–HAM–CYCLE$^{(G)}$:

1 for each $w \in V$

2 If CHECK–HAM–CYCLE$^{(G)}$ returns true then

3 print the Hamiltonian cycle of G.

The above algorithm iterates $O(V)$ times and calls the algorithm CHECK–HAM–CYCLE$^{(G)}$ once per iteration. The total running time of the algorithm will be the product of V and polynomial time which is still a polynomial–time (degree of polynomial will be increased by 1).

Therefore, if HAM–CYCLE $\in P$ then the problem of listing the vertices of a Hamiltonian cycle in order is solvable in polynomial–time.

34.2-5

Show that any language in NP can be decided by an algorithm running in time $2^{O(n^k)}$ for some constant k.

Any language *L* is decidable by an algorithm '*A*' only when

- If there exist an item '*a*' which belongs to '*A*' ($a \in A$) and *A (a) = 1.*
- If x $\notin$ L then A(a) = 0

As per the definition of the NP class, a language *L* belongs to NP only if :

- There exist a two-input polynomial time algorithm *A* and a constant *c.*
- The Language $L = \{a \in \langle 0,1 \rangle\}^*$
- Let y be a certificate with |y| = O(|a|^c) such that *A(a, b) =1*

If the above constraints are satisfied, then the algorithm *A* verifies the language *L* in polynomial time.

Design an algorithm A', for a language L in NP to decide L in times of $2^{O(n^k)}$ for some constant k.

Algorithm A':

For different values of ' *y*'

If A(a, b) = = 1

return 1

else

return 0

Calculating the running time:

- Number of possible y are $2^{|y|}$
- Here, |y| = O(|x|^c)
- Therefore, numbers of possible y are $2^{O(|x|^c)}$.
- For each y, the algorithms runs *A(x, y)* once, in polynomial time of O(|x|^t) for some constant t.
- Therefore, the total running time of the algorithm is

$$2^{O(|x|^c)} O(|x|^t) = 2^{O(n^c)} \left(c_1 n^t\right) = 2^{O(n^c)} \left(2^{\log c_1 + t \log_2 n}\right) = 2^{O(n^k)}$$

34.2-6

A ***hamiltonian path*** in a graph is a simple path that visits every vertex exactly once. Show that the language HAM-PATH $= \{\langle G, u, v \rangle$: there is a hamiltonian path from u to v in graph $G\}$ belongs to NP.

Hampath $\in$ NP
Cerificate: A sequence of vertices y
Verification: check whether y is a Hamiltonian path from u to v in G
Algorithm A (< G, U, V>, < y>)

1. check whether y has n vertices; if not return 0;
2. check whether $y = (y_1, y_2, \ldots\ldots y_n)$ has repeated vertices; if so return 0;
3. check whether $\{y_i, y_{i+1}\} \in E$ for $i = 1, \ldots\ldots n-1$ and whether $\{y_n, y_1\} \in E$ if some of the tests fail then return 0;
4. check whether $y_1 = u$ and $y_n = v$ if not return o; otherwise return 1;

A runs in polynomial time since
Step 1 runs in O (n) steps
Step 2 runs in O (n) steps
Step 3 runs in O (n) steps
Step 4 runs in O (1) steps
So a runs in O (n) steps
It is easy to see A is a correct verification algorithm for HAMPATH

34.3-2

Show that the $\leq_P$ relation is a transitive relation on languages. That is, show that if $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$, then $L_1 \leq_P L_3$.

Consider a polynomial-time reduction $\leq_P$. The polynomial-time reduction $L \leq_p M$ means, "there exists a computable function f : alphabet of L → alphabet of M in such a way that for all $k \in \{0,1\}^*$, $k \in L$ if and only if $f(k) \in M$".

Proving $\leq_P$ relation is a transitive relation:

• Consider two polynomial-time reductions $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$. In other words, there exists a polynomial-time computable reduction functions $f_1 : \{0,1\}^* \to \{0,1\}^*$ and $f_2 : \{0,1\}^* \to \{0,1\}^*$ in such a way that :

$k \in L_1 \Leftrightarrow f_1(k) \in L_2$ ..... (1)

$k \in L_2 \Leftrightarrow f_2(k) \in L_3$ ..... (2)

• Now, define a function $f_3 : \{0,1\}^* \to \{0,1\}^*$ such that, $f_3(x) = (f_2 \circ f_1)(x) = f_2(f_1(x))$.

Now, it is enough to prove that L_1 is polynomial-time reducible to L_3 (i.e. $L_1 \leq_p L_3$). In other words, for all $k \in \{0,1\}^*$, $k \in L_1 \Leftrightarrow f_3(k) \in L_3$.

• Assume that, $k \in L_1$,

o Then, from equation (1), $f_1(k) \in L_2$.

o Since, $f_1(k) \in L_2$, from equation(2), $f_2(f_1(k)) \in L_3$. That is $f_3(k) \in L_3$

Thus, if $k \in L_1$, then $f_3(k) \in L_3$ ..... (3)

• Let $f_3(k) = (f_2 o f_1)(k) = f_2(f_1(k)) \in L_3$.

o Then according to equation (2), $f_1(k) \in L_2$

o Since $f_1(k) \in L_2$, according to equation (1), $k \in L_1$

Thus, if $f_3(k) = f_2(f_1(k)) \in L_3$, then $k \in L_1$ ..... (4)

From equations (3) & (4), $L_1 \leq_p L_3$ that is, $k \in L_1$ if and only if $f_3(k) \in L_3$.

Hence, there exist a polynomial-time reduction from L_1 to L_3 or $L_1 \leq_p L_3$, if $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$.

34.4-2

Show the 3-CNF formula that results when we use the method of Theorem 34.10 on the formula (34.3).

Definition of 3 CNF-SAT: - It is known as the 3-Conjuctive Normal Form Satisfiability.

A literal in a Boolean formula is the occurrence of a variable or its negation. A Boolean formula is in CNF if it is expressed as a conjunction (AND) of clauses each of which is the disjunction (OR) of one or more literals. 3 CNF means that each clause has exactly 3 literals.

For example: $(x_1 \vee \neg x_1 \vee \neg x_3) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

As per the Theorem 34.10

• 3-CNF-SAT ∈ NP [Proof same as SAT]

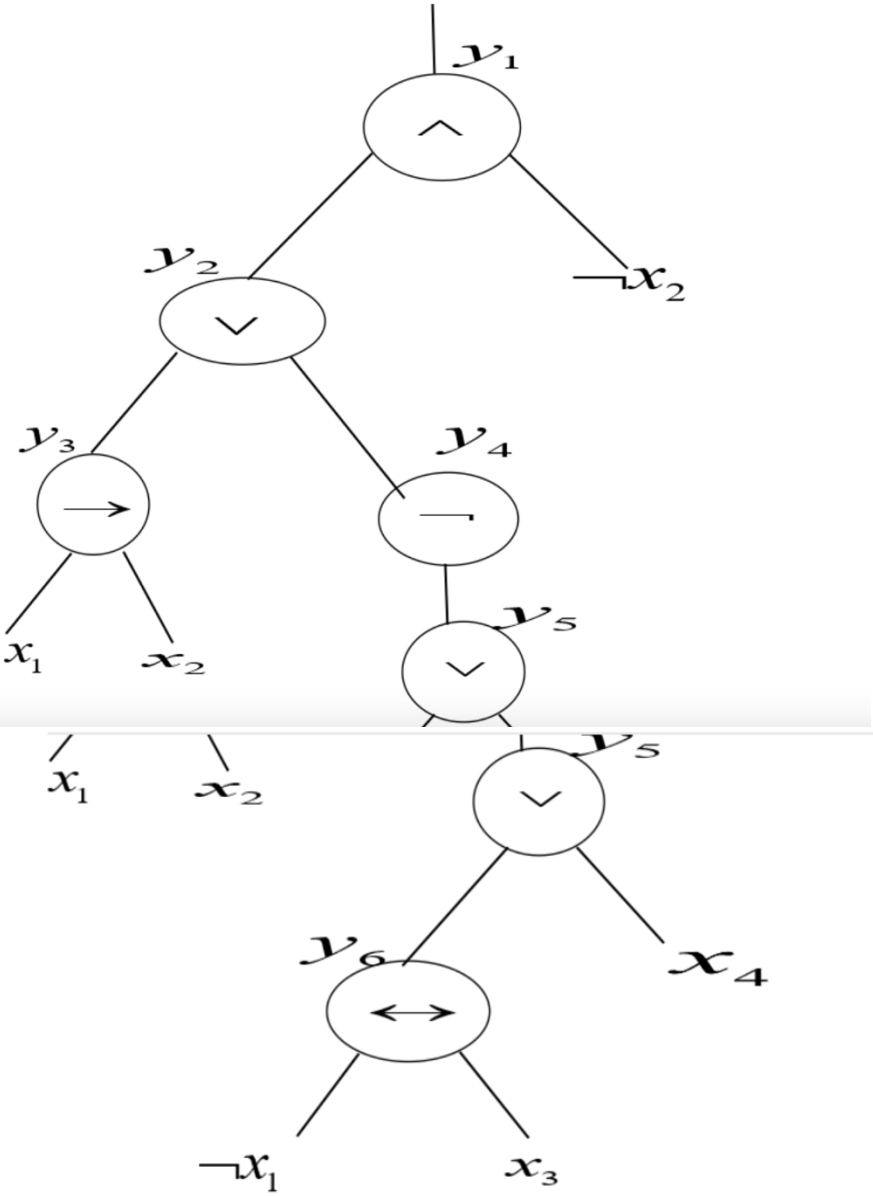
• 3-CNF-SAT ∈ NP-Hard: SAT $\leq_P$ 3-CNF-SAT.

Break the reduction steps into 3 steps as follows:

Step 1:

• Construct a "binary parse tree" for the input formula ϕ. The tree corresponds to the formula ϕ. The tree corresponds to the formula:

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4) \wedge \neg x_2$$



• If the formula contains the AND or OR of several literals, then it is guaranteed that that the internal nodes has just one/ two children by the parenthesis and associativity.

• The binary tree can be interpreted as a circuit for the computation of ϕ . In order to obtain a 3-CNF expression, introduce a variable y_i corresponding to the output of each internal node.

• Rewrite ϕ as AND of the root variable and a conjuction of clauses corresponding to ϕ becomes:-

$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ &\wedge (y_4 \leftrightarrow \neg y_5) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

ϕ' is a conjuction of clauses ϕ'_i each with atmost three literals .

Change ϕ'_i such that each clause has exactly 3 literals connected with OR's

Step 2:

• Change each clause ϕ_i' into CNF. Construct the truth table for each ϕ_i'. Each row of the truth table consists of possible assignment of variables of clause, together with the value of the clause under that assignment. The truth table is as follows:

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

• Using the table entries that evaluate to zero, build a formula in DNF equivalent to ϕ_i'.

The DNF formula equivalent to $\neg\phi_i'$ is:

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$$

• Use Demorgan's law to get the CNF formula:

• Use Demorgan's law to get the CNF formula:

In the above example, convert the clause $\phi_i' = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ into CNF and the

CNF formula is as follows:

$\phi_i'' = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$ which is

equal to the original clause ϕ.

Convert all other formulae ϕ_i' into CNF. Each clause has no more than 3 literals corresponding to two input and one output associated with the nodes of binary parse tree for ϕ .

Step 4/5

Step 3:

• Modify ϕ'' so that all its clauses have exactly 3 distinct literals.

Perform this transformation in several steps.

Modify each clause C_i of ϕ'' after introducing new variables *p* and *q*.

o If C_i has 3 distinct literals: do nothing.

o If C_i has 2 distinct literals: $C_i = (l_1 \vee l_2)$. Replace C_i by $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ which is equivalent to C_i

o If C_i has 1 literal: $C_i = l$. Replace C_i by

$(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$.

Now, show that the 3-CNF formula of ϕ'' is satisfiable, if ϕ is satisfiable .

Examine the 3 reduction steps as follows:

Step 1: The binary tree circuit is another representation of the formula. It will be satisfiable if the formula ϕ' is satisfiable. Apply the induction proof on the number of clauses.

Step 2: Demorgan's law is used to produce a formula which is algebraically equivalent to the original one, that is, ϕ.

Step 3: The new formula is effectively equivalent to the one produced in the Step 2, since a formula algebraically equivalent to that produced in Step 2 is produced by the assignment of the new variables *p* and *q*.

34.4-3

Professor Jagger proposes to show that SAT $\leq_P$ 3-CNF-SAT by using only the truth-table technique in the proof of Theorem 34.10, and not the other steps. That is, the professor proposes to take the boolean formula ϕ, form a truth table for its variables, derive from the truth table a formula in 3-DNF that is equivalent to $\neg\phi$, and then negate and apply DeMorgan's laws to produce a 3-CNF formula equivalent to ϕ. Show that this strategy does not yield a polynomial-time reduction.

The 3-SAT is a satisfiability problem where the Boolean formula has only 3 literals or variables ORed to each other in each clause that are ANDed. The output of each of the clauses should give the output of the whole problem as 1.

It means that we should choose the values of the literals or variables used in the formula so that the output is 1 then only the formula is satisfiable.

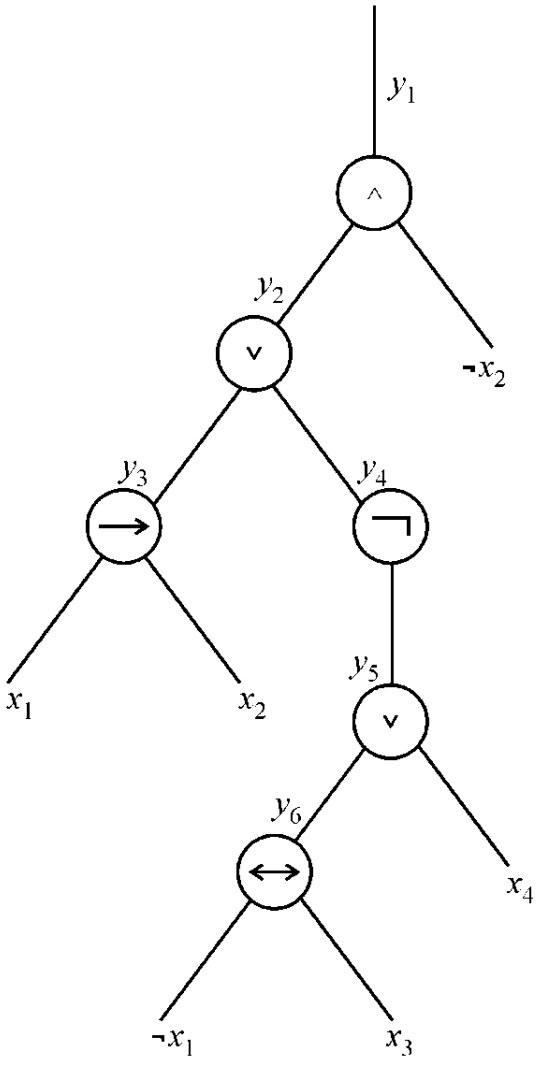
The 3-CNF SAT is a SAT problem with 3 such literals as variables, and 2 CNF SAT is a SAT problem with two literals as variables.

For the proof of 3-CNF SAT to be NPC, consider the following 3-CNF SAT provided in equation form by using 3 literals x_1, x_2, x_3 as follows:

$$\phi = \left((x_1 \to x_2) \vee \neg\left((\neg x_1 \leftrightarrow x_3) \vee x_4\right) \wedge \neg x_2\right)$$

Now, for each clause we need to draw a parse tree and find the corresponding truth table as for example the first clause is been described as follows.

We draw a parse tree provided in figure Fig1 and treat the connectives as nodes and the leaf nodes as the literals.



Corresponding to the above parse tree the result ϕ' of the expression ϕ is derived and represented as follows:

$$\begin{aligned}\phi' = {} & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

Step 2/2

The truth thus derived for the 1st clause that is the truth table for $y_1 \leftrightarrow (y_2 \wedge \neg x_2)$ is as follows:

y_1	y_2	x_2	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

We know that the length of the parse tree is the total height traversed, that is, of the form $h = 2^{\log n}$.

Here *n* is the number of leaves. Now, this is of the form $O(2^k)$ which is a non-polynomial form.

So, to find out the total value of the formula we need to perform the same task for every clause. This results in a non-polynomial degree.

The 3 CNF form is:

$$\phi = (\neg y_1 \vee \neg y_2 \vee x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \neg y_2 \vee x_2)$$

Hence, by taking the values of the truth table the number of values increases creating a non-polynomial reduction. So, this strategy does not yield a polynomial time reduction.

34.4-5

Show that the problem of determining the satisfiability of boolean formulas in disjunctive normal form is polynomial-time solvable.

Disjunctive Normal Form (DNF):

A Boolean formula that is composed of OR of clause of ANDs is known as DNF.

The Boolean formula of the form DNF is satisfiable if any of its clause can be evaluated as 1.

For some variable x such that the clause contains $x \wedge \neg x$ irrespective of the Boolean value x will be evaluated to 0. Otherwise, for some assignment of values 0 and 1 the clause will be evaluated to 1.

The following is the algorithm to determine the satisfiability of Boolean formula in DNF:

Algorithm DNF–SAT $(\langle\phi\rangle)$ **:**

1 for each clause C_i of $\langle\phi\rangle$

2 for variable x such that $x \wedge \neg x$

3 if $x \wedge \neg x$ exists in the clause C_i of $\langle\phi\rangle$

4 retrun false

5 else

6 retrun true

7 end for

8 end for

Running Time Analysis:

- Consider m be the total number of clauses and each clause has at most k (literals) variables in ϕ.
- The loop in the line 1 and its block in the lines 2–8 runs at most m times.
- The loop in the line 3 and its block in the lines 2–7 runs at most k times.
- The block in the lines 3–6 takes at most k time as the existence of $x \wedge \neg x$ in the clause C_i will take k steps.
- So the running time of the algorithm is $m \times k \times k = mk^2$.
- Hence the time complexity of the algorithm DNF–SAT $(\langle\phi\rangle)$ is $O(mk^2)$.

Therefore, the problem of determining whether the satisfiability of Boolean formulas in DNF form is polynomial-time solvable.

34.4-6

Suppose that someone gives you a polynomial-time algorithm to decide formula satisfiability. Describe how to use this algorithm to find satisfying assignments in polynomial time.

The language for formula satisfiability problem is SAT= $\{(\phi) : \phi \text{ is a satisfiable boolean formula}\}$.

Let A denote the polynomial time algorithm which returns 1 if the input x is a satisfiable formula and 0 otherwise.

In addition, assume ϕ has at most n variables denoted as $\{y_1, y_2, \ldots y_n\}$.

The following procedure can be used to find satisfying assignments in polynomial time:

- Apply the polynomial time algorithm A on the Boolean formula ϕ to determine whether ϕ is satisfiable or not. If it is not satisfiable return 0 else continue.
- Initialize a variable, i =1.
- Let the variable $y_i = 0$ then the condition that ϕ is a Boolean formula with n-i variables is satisfied.
- Apply the algorithm A to check the satisfiability that is, there must be an assignment of y_i that satisfies ϕ.
- Replace the variable y_i in Boolean formula ϕ by the recorded value from the previous step and then increment the variable i by 1, repeat the previous step until i = n.
- Return the recorded assignments for all the variables $\{y_1, y_2, \ldots y_n\}$.
- Since, the algorithm A runs in polynomial time, so the above procedure runs in polynomial time.

34.4-7

Let 2-CNF-SAT be the set of satisfiable boolean formulas in CNF with exactly 2 literals per clause. Show that 2-CNF-SAT $\in$ P. Make your algorithm as efficient as possible. (*Hint:* Observe that $x \vee y$ is equivalent to $\neg x \rightarrow y$. Reduce 2-CNF-SAT to an efficiently solvable problem on a directed graph.)

Show that 2 – CNF is solvable in Polynomial time. Assume W.L.O.G. that each clause contains exactly 2 literals. Following the hind we construct a directed graph $G=(V,E)$ as follows

- Let $X_{0,\ldots}x_n$ be the variables in the formula. There are two vertices vi and $\overline{vi}$ for each x_i. The vertex V_i corresponds to x_i and $\overline{v}_i$ corresponds to $\neg x_i$
- for each clause we construct two edges for example, given for $x_i \vee \neg x_j$ we create $(V_j, V_i)\ (\overline{V}_i \overline{V}_j)$

We claim that this formula is satisfiable if and only if no pair of complimentary literals are in the same strongly connected of G. if there are paths from U to V and vice versa. Then in any thruth assignment the corresponding literals. Must have the same value since a path is chain of implications.

Conversely suppose no pair of complementary literals are in the same strongly connected component. Consider the dag obtained by contracting each strongly connected component to a single vertex This Dag induces a partial order, which we then extend to a total order using to pological sort. For each x_i. If the component of Vi proceeds the component of $\overline{V}_i$ set $v_i = 0$ else set $x_i = 1$. We claim that this is valid truth assignment ie. That

(i) All Literals in the same component are assigned the same values and

(ii) If a component B is reachable from a then A, B cannot be assigned 1, 0.

We first prove (i). Assume for the contrary that two literals L_1 and L_2 are in the same strongly connected component S but the strongly connected component containing $\neg$7Li proceeds S in the total order and the component containing $\neg L_2$ is proceeded by S since L_1 and L_2 are in the same. Component $L_1 \rightarrow L_x$ and $L_2 \rightarrow L_1$ and $(\neg L_1 \vee L_2)$ can be obtained. Hence there must be a path from $\neg L_2$ to $\neg L_1$ this contradicts the total order. We prove (ii). Assume for contradiction that there are two connected components a and b such that B is reachable from A, but our algorithm assigns 1 and 0 to A and B. Let L_a and L_b be a literals in A and B respectively. Note that there must be a path from $\neg l_a$ and $\neg l_a$. Let $\overline{B}$ and $\overline{A}$ be the component of $\neg l_a$ and $\neg 7 l_B$ clearly, $\overline{B}$ has value 1 and $\overline{A}$ has value 0. In the total order B preceded $\overline{B}$ and $\overline{A}$ proceeded a this implies that there is a cycle in the total order.

34.5-2

Given an integer $m \times n$ matrix A and an integer m-vector b, the ***0-1 integer-programming problem*** asks whether there exists an integer n-vector x with elements in the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 integer programming is NP-complete. (*Hint:* Reduce from 3-CNF-SAT.)

It is required to show that any 0-1 solution to Ax = b provides a satisfying assignment. Let x be the solution which satisfies the given assignment. The inequalities which arise by the last 2n rows of b guarantee that exactly one of x[m] and x [n + m] is equal to 1. It also means that each x_i is either true or false.

Step 2/2

The 0-1 integer programming problem lies in the class NP because the verification of the solution takes linear time.

The i^{th} row of b is equal to -1, which means at least one literal is true. So, 0-1 LP is NP-complete.

34-2 Bonnie and Clyde

Bonnie and Clyde have just robbed a bank. They have a bag of money and want to divide it up. For each of the following scenarios, either give a polynomial-time algorithm, or prove that the problem is NP-complete. The input in each case is a list of the n items in the bag, along with the value of each.

a. The bag contains n coins, but only 2 different denominations: some coins are worth x dollars, and some are worth y dollars. Bonnie and Clyde wish to divide the money exactly evenly.

b. The bag contains n coins, with an arbitrary number of different denominations, but each denomination is a nonnegative integer power of 2, i.e., the possible denominations are 1 dollar, 2 dollars, 4 dollars, etc. Bonnie and Clyde wish to divide the money exactly evenly.

c. The bag contains n checks, which are, in an amazing coincidence, made out to "Bonnie or Clyde." They wish to divide the checks so that they each get the exact same amount of money.

d. The bag contains n checks as in part (c), but this time Bonnie and Clyde are willing to accept a split in which the difference is no larger than 100 dollars.

Bonnie and Clyde

A problem is said to be in a set of NP-Complete if that can be solved in polynomial time, here NP is abbreviation of non-deterministic polynomial time. For example if a language named L is contained in the set NP then there exists an algorithm A for a constant c then

$$L=\left\{x\in\{0,1\}^{*}:\text{there exists a certificate } y \text{ with} |y|=\mathrm{O}\left(|x|^{c}\right)\text{such that } A(x,y)==1\right\}$$

Any problem that is convertible to the decision problem will surely be the polynomial time .

Step 2/7

(a)

Consider that there are m coins whose worth is x dollars each and rest $n-m$ coins worth y dollars each. So the sum total of the worth of all the coins is:

$S=mx+(n-m)y$ dollars.

So for equal division of the total worth Bonnie and Clyde must take $S/2$ dollars each. The division of coins can be done in a manner so that each of them have equal amount so for division assume that either Bonnie or Clyde start taking the coins and take exactly half worth coins.

Assume that Bonnie first start to take the coins and he is free to take any of the coin that is either x worth or y worth and any number of coins whose total worth is exactly $S/2$ dollars.

If Bonnie takes p number of coins of dollar x where $0\le p\le m$ then to check how many coins of y dollars Bonnie should take that is to check if $px\le S/2$ and $\left(\frac{S}{2}-px\right)\Big/y\le(n-m)$.

Bonnie can take $0,1,2,\ldots,m$ coins of value x dollars so there are maximum $m+1$ possibilities to check the worth of remaining coins, because Bonnie can take at most m coins. For every possibility, perform a fixed number of operations, there are a total of polynomial number of possibilities. Constant number of operations for polynomial number of possibilities cause this algorithm to be a polynomial time solvable algorithm.

Hence, the above is the polynomial time method to divide the coins exactly amond them.

(b)

The following method is deployed to do the required division of coins.

1. Arrange all the *n* coins in the non-increasing order of their denominations that is $2^0, 2^1, 2^2, \cdots$. Firstly give Bonnie the coin with greatest worth or denomination after that give Clyde the coins the sum of whose worth is same as the coin given to Bonnie.

2. Now the worth of remaining coins will be greater than or equal to the sum of worth of coins which are with Bonnie and Clyde because the denominations of coins are raised to power of two.

3. Now from remaining coins give the coin which has maximum worth to Bonnie and again give same worth coins to Clyde and repeat this process until all the coins are divided.

4. In this process if a coin is given to Bonnie which is of cost *p* and the sum of worth of rest of the coins is less than *p* then coins can't be divided equally between them else it is possible.

This process requires polynomial number of steps and each step takes constant steps each. So, the total asymptotic time complexity of this procedure is polynomial in nature.

Hence, the above is the polynomial time method to divide the coins exactly amond them.

Step 4/7

(c)

This problem is solved by following steps:

• The problem is in **NP** or not. If it is, then go to next step.

• Is every problem in NP is reducible to this problem in polynomial time.

• If both of these are true, then the problem is **NP-COMPLETE**.

1. First check if the problem is in NP.

Once the division of checks is done, the sum of the checks should be equal to total sum. This is done in two steps, first use list of items and the value of each to find the total value as follows:

$$Total = n_1 v_1 + n_2 v_2 + \ldots\ldots n_n v_n$$

Next calculate the value of checks that one of them gets. If that value is equal to $\frac{Total}{2}$, then the algorithm is verified. So, the problem definitely belongs to **NP.**

2. Check if every problem in NP is reducible to this problem in polynomial time.

Consider the SET-PARTITION problem which states that it takes polynomial time to find out if there is a subset in a bigger set such that the sum of the elements in the subset is equal to a given integer *k.*

To reduce this problem to the current problem, consider the set to be the set of all checks in the bag and the integer $k = \frac{Total}{2}$. This reduction takes polynomial time.

Infact, SET-PARTITION problem is NP-COMPLETE. It is reducible to the current problem.

From these two steps, it can be said that the problem belongs to NP and every problem in NP is reducible to this problem in polynomial time, therefore, the problem in NP-complete.

Step 6/7

(d)

This problem is solved by following steps:

• The problem is in **NP** or not. If it is, then go to next step.

• Is every problem in NP is reducible to this problem in polynomial time.

• If both of these are true, then the problem is **NP-COMPLETE**.

1. First check if the problem is in NP.

Once the division of checks is done, the difference between the total with Bonnie and the total with Clyde must be less than equal to 100.

This is done in two steps, first use list of items and the value of each to find the total value for both Bonnie and Clyde.

Next calculate the difference is less than equal to 100 or not as follows:

$$\left|\text{Bonnie}_{total} - \text{Clyde}_{total}\right| \leq 100$$

If it is true, then the algorithm is verified. So, the problem definitely belongs to **NP.**

2. Check if every problem in NP is reducible to this problem in polynomial time.

Consider the SET-PARTITION problem which states that it takes polynomial time to find out if there is a subset in a bigger set such that the sum of the elements in the subset is equal to a given integer *k.*

To reduce this problem to the current problem, consider the set to be the set of all checks in the bag and then check for each value of x such that $0 \le x \le 100, \forall x \in Z$, the value of $\text{SET-PARTITION}\left(S, \frac{S}{2} - x\right) = \text{SET-PARTITION}\left(S, \frac{S}{2} + x\right)$ is true. This reduction takes polynomial time as it requires running the SET-PARTITION algorithm for 200 times.

Infact, SET-PARTITION problem is NP-COMPLETE. It is reducible to the current problem.

From these two steps, it can be said that the problem belongs to NP and every problem in NP is reducible to this problem in polynomial time, therefore, the problem in NP-complete.