

# More NP-Complete Problems

NP-Hard Problems

Tautology Problem

Node Cover

Knapsack

# Next Steps

- ◆ We can now reduce 3SAT to a large number of problems, either directly or indirectly.
- ◆ Each reduction must be polytime.
- ◆ Usually we focus on length of the output from the transducer, because the construction is easy.
- ◆ But **key issue**: must be polytime.

## Next Steps – (2)

- ◆ Another essential part of an NP-completeness proof is showing the problem is in **NP**.
- ◆ Sometimes, we can only show a problem *NP-hard* = “if the problem is in **P**, then **P** = **NP**,” but the problem may not be in **NP**.

# Example: NP-Hard Problem

- ◆ The *Tautology Problem* is: given a Boolean formula, is it satisfied by **all** truth assignments?
  - ◆ Example:  $x + \neg x + yz$
- ◆ Not obviously in **NP**, but it's complement is.
  - ◆ Guess a truth assignment; accept if that assignment doesn't satisfy the formula.

# Key Point Regarding Tautology

- ◆ An NTM can guess a truth assignment and decide whether formula  $F$  is satisfied by that assignment in polytime.
- ◆ But if the NTM accepts when it guesses a satisfying assignment, it will accept  $F$  whenever  $F$  is in SAT, not Tautology.

# Co-NP

- ◆ A problem/language whose complement is in **NP** is said to be in *Co-NP*.
- ◆ **Note**: **P** is closed under complementation.
- ◆ Thus, **P**  $\subseteq$  Co-**NP**.
- ◆ Also, if **P** = **NP**, then **P** = **NP** = Co-**NP**.

# Tautology is NP-Hard

- ◆ While we can't prove Tautology is in **NP**, we can prove it is NP-hard.
- ◆ Suppose we had a polytime algorithm for Tautology.
- ◆ Take any Boolean formula  $F$  and convert it to  $\neg(F)$ .
  - ◆ Obviously linear time.

# Tautology is NP-Hard – (2)

- ◆ F is satisfiable if and only  $\neg(F)$  is **not** a tautology.
- ◆ Use the hypothetical polytime algorithm for Tautology to test if  $\neg(F)$  is a tautology.
- ◆ Say “yes, F is in SAT” if  $\neg(F)$  is not a tautology and say “no” otherwise.
- ◆ Then SAT would be in **P**, and **P = NP**.



# Historical Comments

- ◆ There were actually two notions of “NP-complete” that differ subtly.
  - ◆ And only if  $P \neq NP$ .
- ◆ Steve Cook, in his 1970 paper, was really concerned with the question “why is Tautology hard?”
  - ◆ **Remember**: theorems are really logical tautologies.

## History – (2)

- ◆ Cook used “if problem  $X$  is in  $\mathbf{P}$ , then  $\mathbf{P} = \mathbf{NP}$ ” as the definition of “ $X$  is NP-hard.”
  - ◆ Today called *Cook completeness*.
- ◆ In 1972, Richard Karp wrote a paper showing many of the key problems in Operations Research to be NP-complete.

## History – (3)

- ◆ Karp's paper moved "NP-completeness" from a concept about theorem proving to an essential for any study of algorithms.
- ◆ But Karp used the definition of NP-completeness "exists a polytime reduction," as we have.
  - ◆ Called *Karp completeness*.

# Cook Vs. Karp Completeness

- ◆ In practice, there is very little difference.
- ◆ **Biggest difference**: for Tautology, Cook lets us flip the answer after a polytime reduction.
- ◆ In principle, Cook completeness could be much more powerful, or (if **P = NP**) exactly the same.

# Cook Vs. Karp – (2)

- ◆ But there is one important reason we prefer Karp-completeness.
- ◆ Suppose I had an algorithm for some NP-complete problem that ran in time  $O(n^{\log n})$ .
  - ◆ A function that is bigger than any polynomial, yet smaller than the exponentials like  $2^n$ .

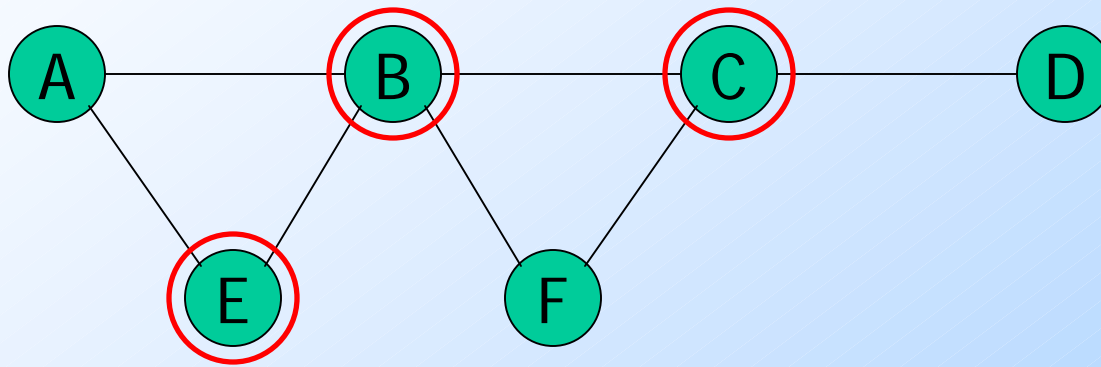
## Cook Vs. Karp – (3)

- ◆ If “NP-complete is Karp-completeness, I can conclude that all of **NP** can be solved in time  $O(n^{f(n)})$ , where  $f(n)$  is some function of the form  $c \log^k n$ .
  - ◆ Still faster than any exponential, and faster than we have a right to expect.
- ◆ But if I use Cook-completeness, I cannot say anything of this type.

# The Node Cover Problem

- ◆ Given a graph  $G$ , we say  $N$  is a *node cover* for  $G$  if every edge of  $G$  has at least one end in  $N$ .
- ◆ The problem Node Cover is: given a graph  $G$  and a “budget”  $k$ , does  $G$  have a node cover of  $k$  or fewer nodes?

# Example: Node Cover



One possible node cover  
of size 3: {B, C, E}

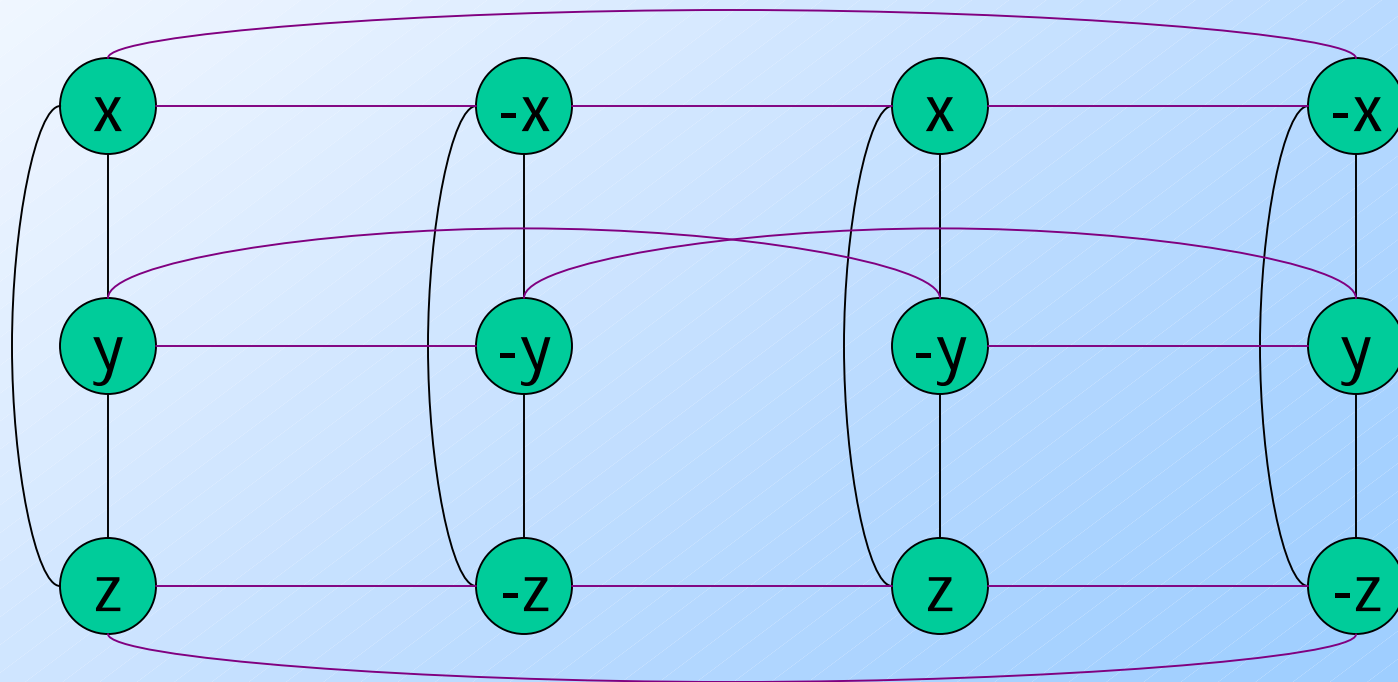


# NP-Completeness of Node Cover

- ◆ Reduction from 3SAT.
- ◆ For each clause  $(X+Y+Z)$  construct a “column” of three nodes, all connected by *vertical* edges.
- ◆ Add a *horizontal* edge between nodes that represent any variable and its negation.
- ◆ Budget = twice the number of clauses.

# Example: The Reduction to Node Cover

$$(x + y + z)(-x + -y + -z)(x + -y + z)(-x + y + -z)$$



Budget  
= 8

## Example: Reduction – (2)

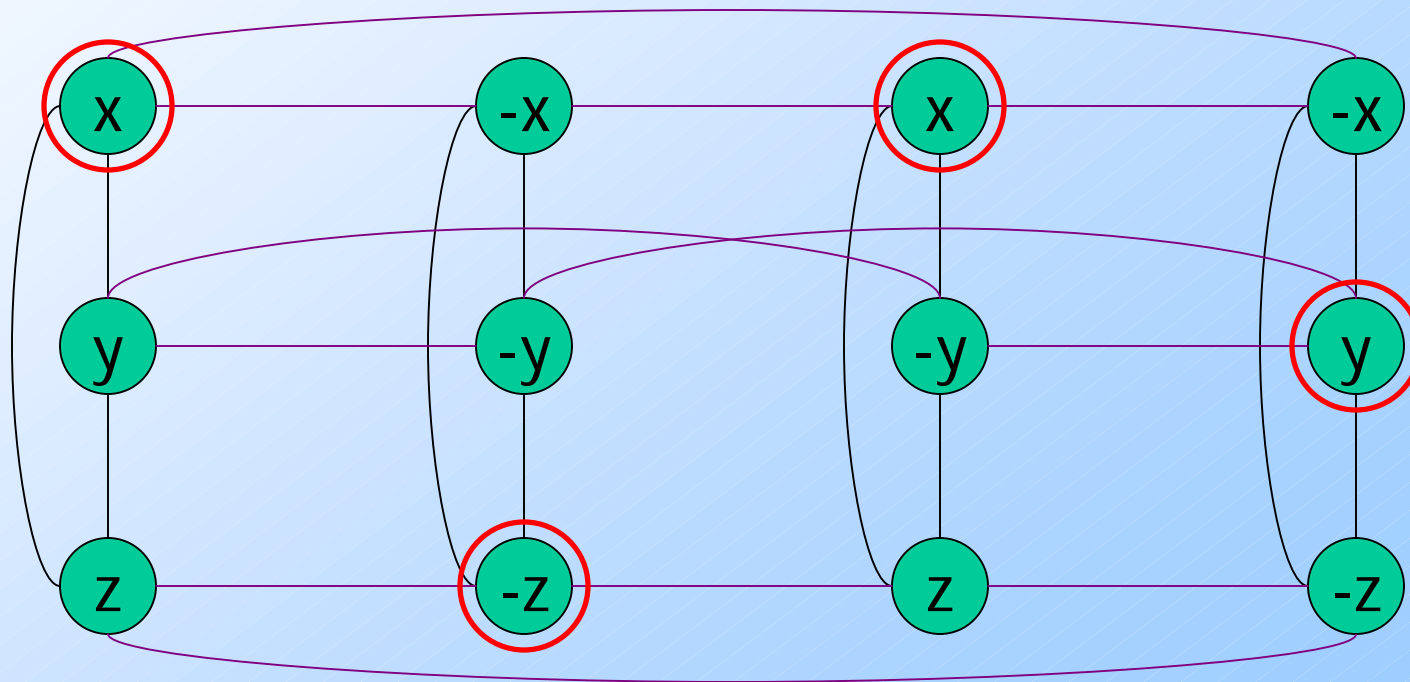
- ◆ A node cover must have at least two nodes from every column, or some vertical edge is not covered.
- ◆ Since the budget is twice the number of columns, there must be exactly two nodes in the cover from each column.
- ◆ Satisfying assignment corresponds to the node in each column **not** selected.

## Example: Reduction – (3)

$$(x + y + z)(-x + -y + -z)(x + -y + z)(-x + y + -z)$$

Truth assignment:  $x = y = T$ ;  $z = F$

Pick a true node in each column

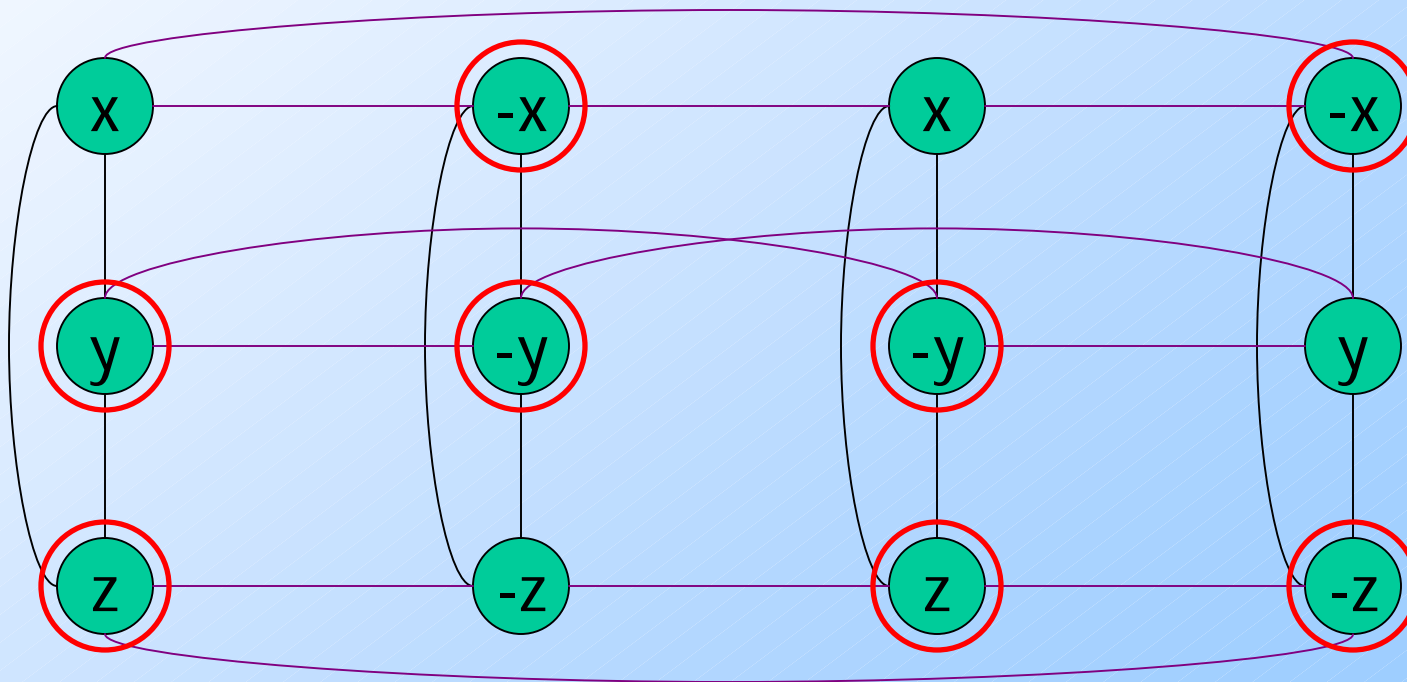


## Example: Reduction – (4)

$$(x + y + z)(-x + -y + -z)(x + -y + z)(-x + y + -z)$$

Truth assignment:  $x = y = T$ ;  $z = F$

The other nodes form a node cover



# Proof That the Reduction Works

- ◆ The reduction is clearly polytime.
- ◆ Need to show:
  - ◆ If we construct from 3SAT instance  $F$  a graph  $G$  and a budget  $k$ , then  $G$  has a node cover of size  $k$  if and only if  $F$  is satisfiable.

## Proof: If

- ◆ Suppose we have a satisfying assignment  $A$  for  $F$ .
- ◆ For each clause of  $F$ , pick one of its three literals that  $A$  makes true.
- ◆ Put in the node cover the two nodes for that clause that do **not** correspond to the selected literal.
- ◆ Total =  $k$  nodes – meets budget.

## Proof: If – (2)

- ◆ The selected nodes cover all vertical edges.
  - ◆ Why? Any two nodes for a clause cover the triangle of vertical edges for that clause.
- ◆ Horizontal edges are also covered.
  - ◆ A horizontal edge connects nodes for some  $x$  and  $-x$ .
  - ◆ One is false in  $A$  and therefore its node must be selected for the node cover.



## Proof: Only If

- ◆ Suppose  $G$  has a node cover with at most  $k$  nodes.
- ◆ One node cannot cover the vertical edges of any column, so each column has exactly 2 nodes in the cover.
- ◆ Construct a satisfying assignment for  $F$  by making true the literal for any node not in the node cover.

## Proof: Only If – (2)

- ◆ **Worry:** What if there are unselected nodes corresponding to both  $x$  and  $-x$ ?
  - ◆ Then we would not have a truth assignment.
- ◆ But there is a horizontal edge between these nodes.
- ◆ Thus, at least one is in the node cover.

# Optimization Problems

- ◆ NP-complete problems are always yes/no questions.
- ◆ In practice, we tend to want to solve *optimization problems*, where our task is to minimize (or maximize) a parameter subject to some constraints.

# Example: Optimization Problem

- ◆ People who care about node covers would ask:
  - ◆ Given this graph, what is the smallest number of nodes I can pick to form a node cover?
- ◆ If I can solve that problem in polytime, then I can solve the yes/no version.

## Example – Continued

- ◆ **Polytime algorithm:** given graph  $G$  and budget  $k$ , solve the optimization problem for  $G$ .
- ◆ If the smallest node cover for  $G$  is of size  $k$  or less, answer “yes”; otherwise answer “no.”

# Optimization Problems – (2)

- ◆ Optimization problems are never, strictly speaking, in **NP**.
  - ◆ They are not yes/no.
- ◆ But there is a Cook reduction from the yes/no version to the optimization version.

# Optimization Problems – (3)

- ◆ That is enough to show that if the optimization version of an NP-complete problem can be solved in polytime, then **P = NP**.
  - ◆ A strong argument that you cannot solve the optimization version of an NP-complete problem in polytime.

# The Knapsack Problem

- ◆ We shall prove NP-complete a version of Knapsack with a budget:
  - ◆ Given a list  $L$  of integers and a budget  $k$ , is there a subset of  $L$  whose sum is exactly  $k$ ?
- ◆ Later, we'll reduce this version of Knapsack to our earlier one: given an integer list  $L$ , can we divide it into two equal parts?



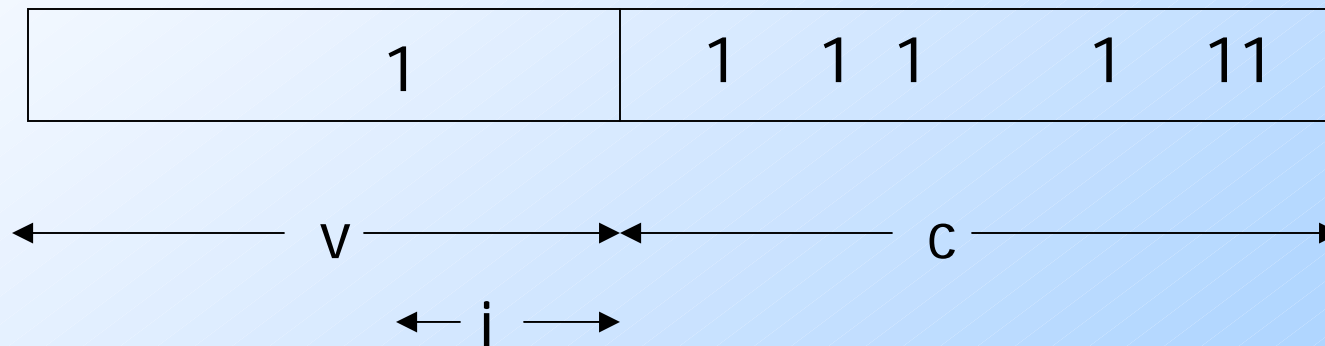
# Knapsack is in **NP**

- ◆ Guess a subset of the list  $L$ .
- ◆ Add 'em up.
- ◆ Accept if the sum is  $k$ .

# Polytime Reduction of 3SAT to Knapsack

- ◆ Given 3SAT instance  $F$ , we need to construct a list  $L$  and a budget  $k$ .
- ◆ Suppose  $F$  has  $c$  clauses and  $v$  variables.
- ◆  $L$  will have base-32 integers of length  $c+v$ , and there will be  $3c+2v$  of them.

# Picture of Integers for Literals

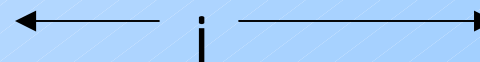
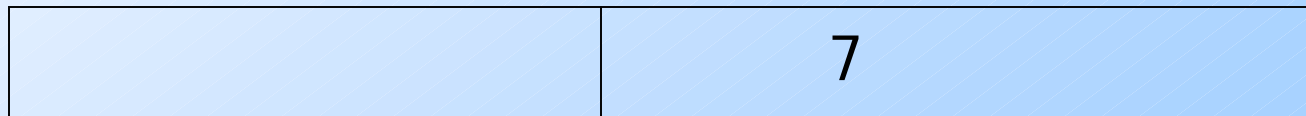
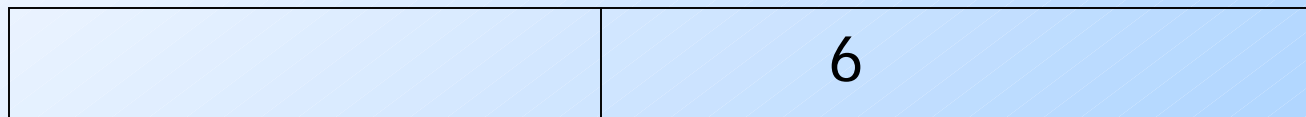
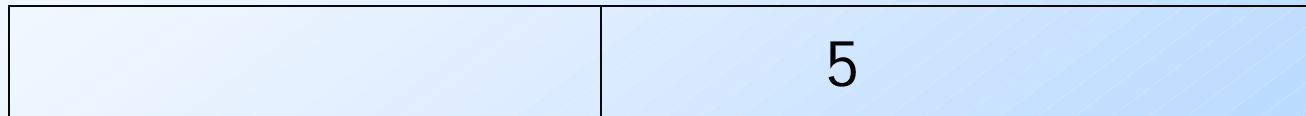


1 in  $i$ -th position  
if this integer is  
for  $x_i$  or  $-x_i$ .

1's in all positions  
such that this literal  
makes the clause true.

All other positions are 0.

# Pictures of Integers for Clauses



For the  $i$ -th clause

# Example: Base-32 Integers

$$(x + y + z)(x + -y + -z)$$

- ◆  $c = 2; v = 3$ .
- ◆ Assume  $x, y, z$  are variables 1, 2, 3, respectively.
- ◆ Clauses are 1, 2 in order given.

Example:  $(x + y + z)(x + -y + -z)$

◆ For x: 00111

◆ For -x: 00100

◆ For y: 01001

◆ For -y: 01010

◆ For z: 10001

◆ For -z: 10010

◆ For first clause:

00005, 00006,  
00007

◆ For second clause:

00050, 00060,  
00070

# The Budget

- ◆  $k = 8(1 + 32 + 32^2 + \dots + 32^{c-1}) + 32^c(1 + 32 + 32^2 + \dots + 32^{v-1})$
- ◆ That is, 8 for the position of each clause and 1 for the position of each variable.
- ◆ **Key Point:** Base-32 is high enough that there can be no carries between positions.

## Key Point: Details

- ◆ Among all the integers, the sum of digits in the position for a variable is 2.
- ◆ And for a clause, it is  $1+1+1+5+6+7 = 21$ .
  - ◆ 1's for the three literals in the clause; 5, 6, and 7 for the integers for that clause.
- ◆ Thus, the budget must be satisfied on a digit-by-digit basis.



## Key Point: Concluded

- ◆ Thus, if a set of integers matches the budget, it must include exactly one of the integers for  $x$  and  $-x$ .
- ◆ For each clause, at least one of the integers for literals must have a 1 there, so we can choose either 5, 6, or 7 to make 8 in that position.

# Proof the Reduction Works

- ◆ Each integer can be constructed from the 3SAT instance  $F$  in time proportional to its length.
  - ◆ Thus, reduction is  $O(n^2)$ .
- ◆ If  $F$  is satisfiable, take a satisfying assignment  $A$ .
- ◆ Pick integers for those literals that  $A$  makes true.

## Proof the Reduction Works – (2)

- ◆ The selected integers sum to between 1 and 3 in the digit for each clause.
- ◆ For each clause, choose the integer with 5, 6, or 7 in that digit to make a sum of 8.
- ◆ These selected integers sum to exactly the budget.

## Proof: Converse

- ◆ We must also show that a sum of integers equal to the budget  $k$  implies  $F$  is satisfiable.
- ◆ In each digit for a variable  $x$ , either the integer for  $x$  or the digit for  $-x$ , but not both is selected.
  - ◆ let truth assignment  $A$  make this literal true.

## Proof: Converse – (2)

- ◆ In the digits for the clauses, a sum of 8 can only be achieved if among the integers for the variables, there is at least one 1 in that digit.
- ◆ Thus, truth assignment A makes each clause true, so it satisfies F.

# The *Partition-Knapsack Problem*

- ◆ This problem is what we originally referred to as “knapsack.”
- ◆ Given a list of integers  $L$ , can we partition it into two disjoint sets whose sums are equal?
- ◆ Partition-Knapsack is NP-complete; reduction from Knapsack.

# Reduction of Knapsack to Partition-Knapsack

- ◆ Given instance  $(L, k)$  of Knapsack, compute the sum  $s$  of all the integers in  $L$ .
  - ◆ Linear in input size.
- ◆ Output is  $L$  followed by two integers:  $s$  and  $2k$ .

◆ **Example:**  $L = (3, 4, 5, 6); k = 7$ .

◆ Partition-Knapsack instance =  $(3, 4, 5, 6, 14, 18)$ .

Solution

Solution

# Proof That Reduction Works

- ◆ The sum of all integers in the output instance is  $2(s+k)$ .
  - ◆ Thus, the two partitions must each sum to  $s+k$ .
- ◆ If the input instance has a subset of  $L$  that sums to  $k$ , then pick it plus the integer  $s$  to solve the output instance.



# Proof: Converse

- ◆ Suppose the output instance of Partition-Knapsack has a solution.
- ◆ The integers  $s$  and  $2k$  cannot be in the same partition.
  - ◆ Because their sum is more than half  $2(s+k)$ .
- ◆ Thus, the subset of  $L$  that is in the partition with  $s$  sums to  $k$ .
  - ◆ Thus, it solves the Knapsack instance.