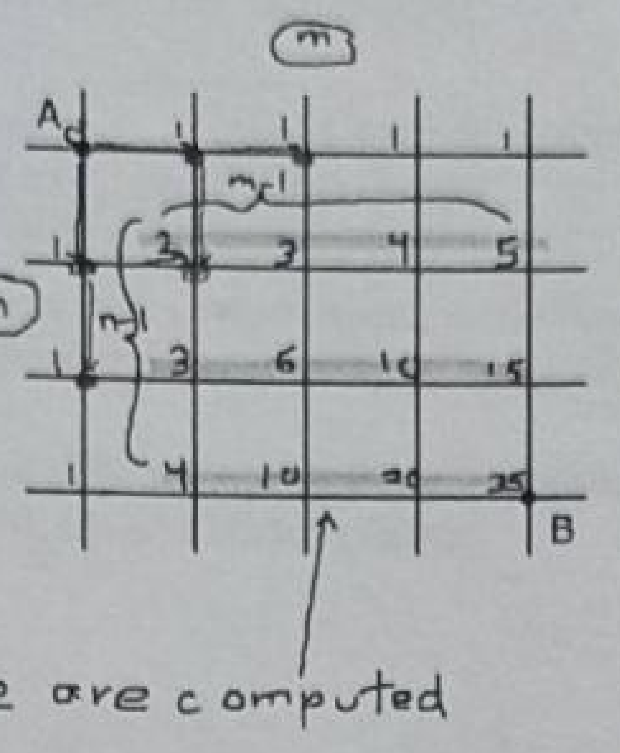
Example 3: Path counting

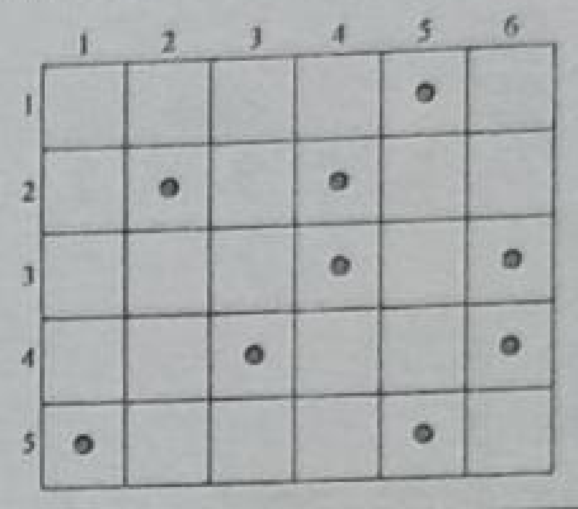
Consider the problem of counting the number of shortest paths from point A to point B in a city with perfectly horizontal streets and vertical avenues



7

Example 4: Coin-collecting by robot

Several coins are placed in cells of an $n \times m$ board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location.



Solution to the coin-collecting problem

- Let F(*i*,*j*) be the largest number of coins the robot can collect and bring to cell (*i*,*j*) in the *i*th row and *j*th column.
- The largest number of coins that can be brought to cell (*i*,*j*):
  - from the left neighbor ?
  - from the neighbor above?
- The recurrence:
  - $F(i, j) = \max\{F(i-1, j),\ F(i, j-1)\} + c_{ij}$ for $1 \le i \le n, 1 \le j \le m$
- where $c_{ij} = 1$ if there is a coin in cell (*i*,*j*), and $c_{ij} = 0$ otherwise
  - $F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le$

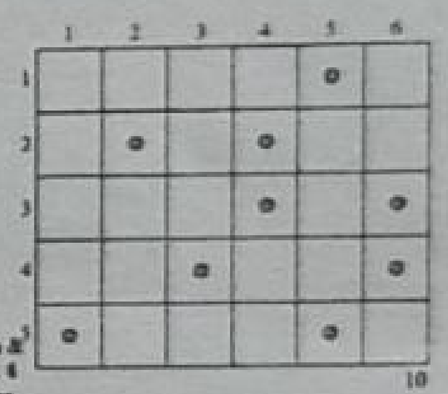
9

Solution to the coin-collecting problem

$F(i, j) = \max\{F(i-1, j),\ F(i, j-1)\} + c_{ij}$ for $1 \le i \le n, 1 \le j \le m$

where $c_{ij} = 1$ if there is a coin in cell (*i*,*j*), and $c_{ij} = 0$ otherwise

$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$.



10

Other examples of DP algorithms

- **Computing a binomial coefficient (# 9, Exercises 8.1)**
- **General case of the change making problem (Sec. 8.1)**
- **Some difficult discrete optimization problems:**
  - knapsack (Sec. 8.2)
  - traveling salesman
- **Constructing an optimal binary search tree (Sec. 8.3)**
- **Warshall's algorithm for transitive closure (Sec. 8.4)**
- **Floyd's algorithm for all-pairs shortest paths (Sec. 8.4)**

11

Knapsack Problem by DP

Given *n* items of

integer weights: $w_1\ w_2 \dots w_n$
values: $v_1\ v_2 \dots v_n$

a knapsack of integer capacity *W*

find most valuable subset of the items that fit into the knapsack

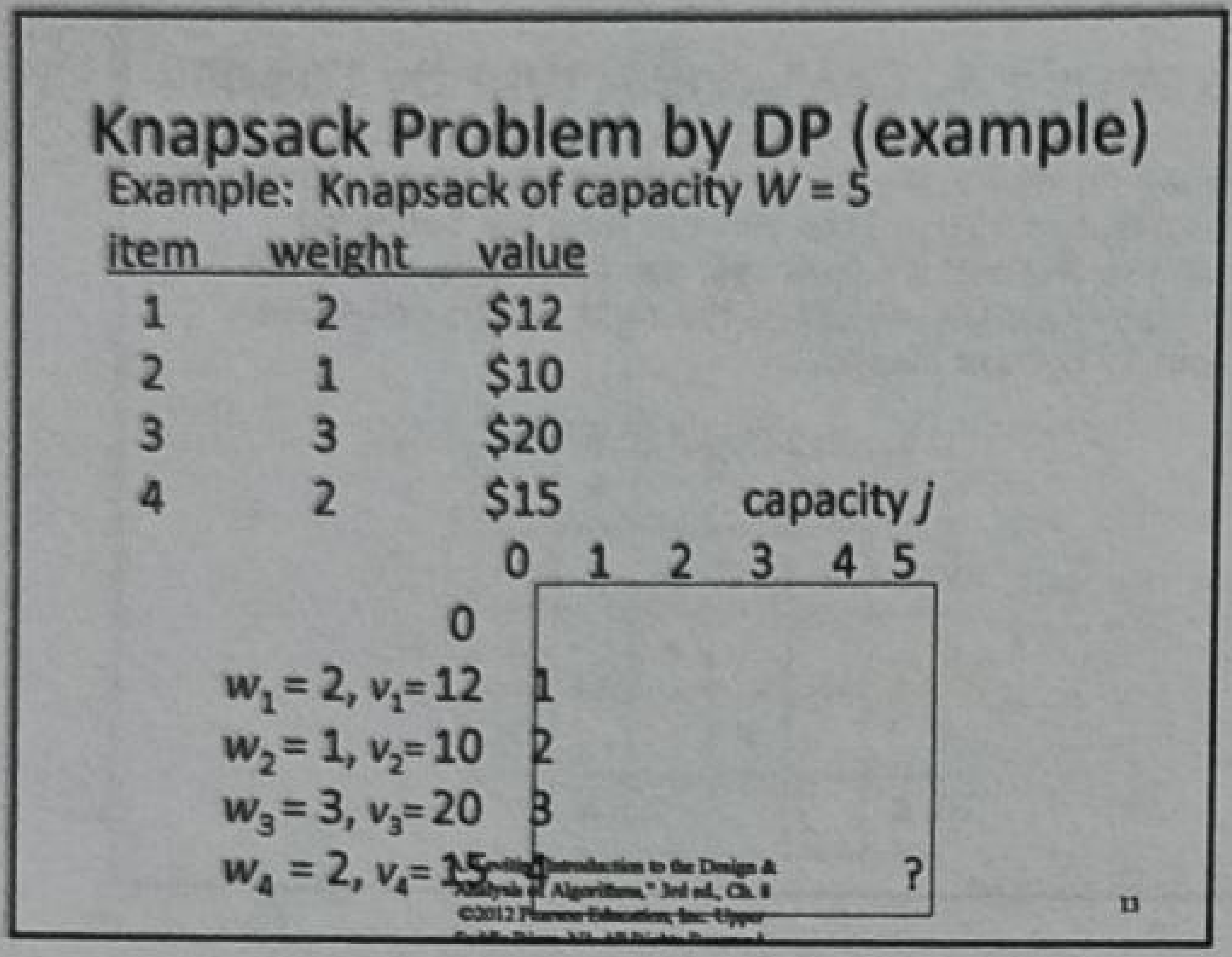
Consider instance defined by first *i* items and capacity *j* ($j \le W$).

Let $V[i,j]$ be optimal value of such instance. Then

$$V[i,j] = \begin{cases} \max\{V[i-1,j],\ v_i + V[i-1,j-w_i]\} & \text{if } j-w_i \ge 0 \\ V[i-1,j] & \text{if } j-w_i < 0 \end{cases}$$

Initial conditions: $V[0,j] = 0$ and $V[i,0] = 0$

12



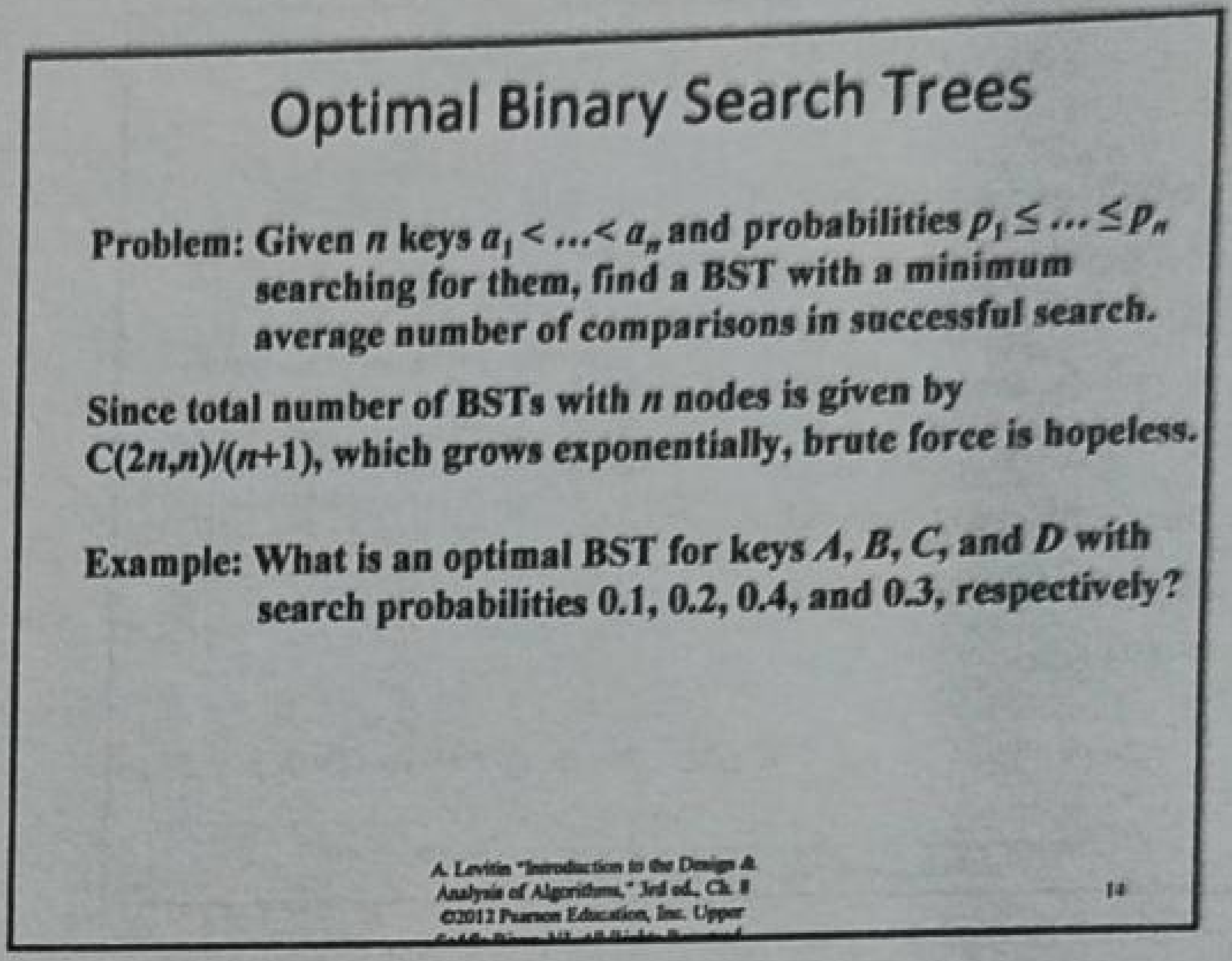
Knapsack Problem by DP (example)

Example: Knapsack of capacity $W = 5$

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity j

		0	1	2	3	4	5
	0						
$w_1 = 2, v_1 = 12$	1						
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						?

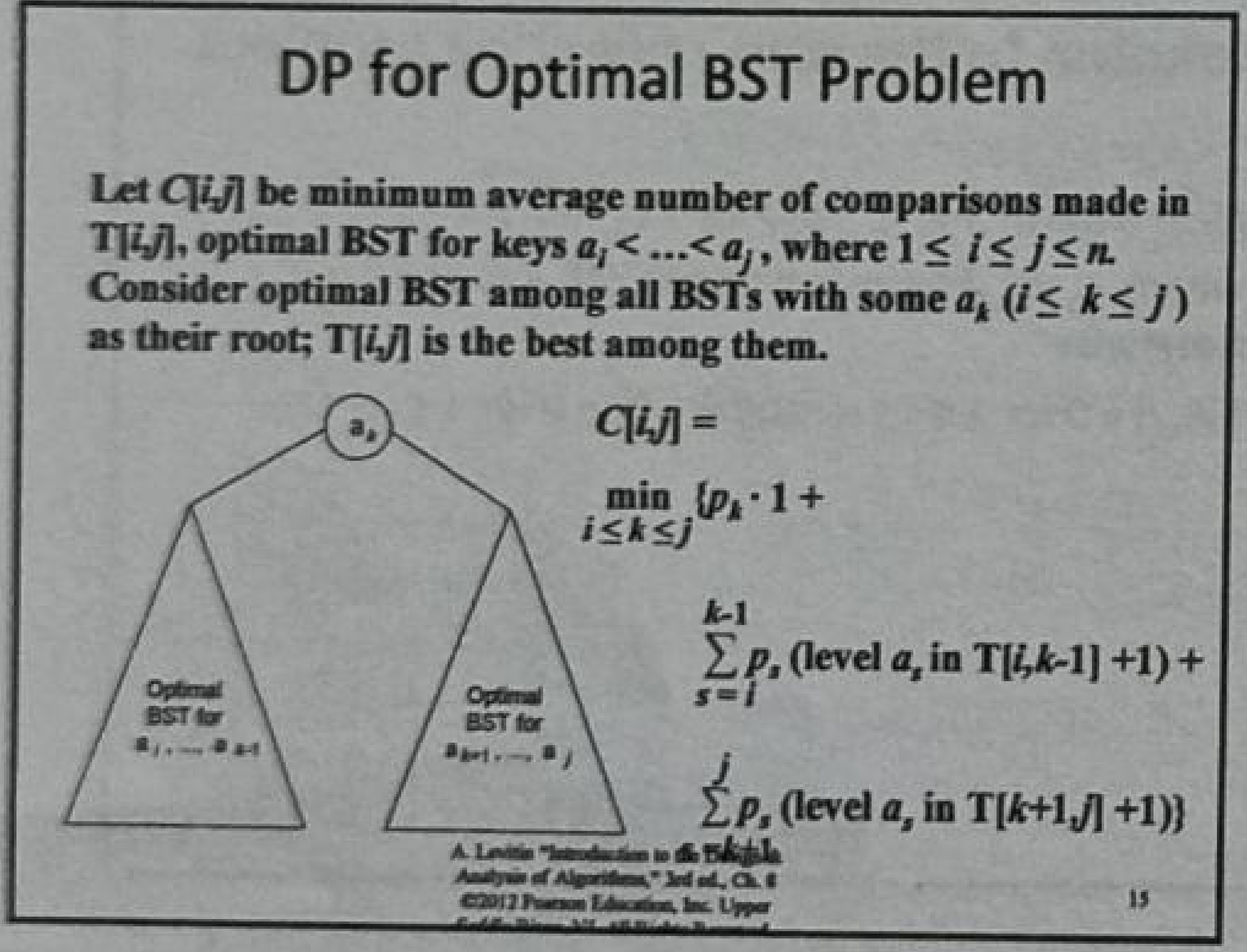


Optimal Binary Search Trees

Problem: Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 \le \ldots \le p_n$ searching for them, find a BST with a minimum average number of comparisons in successful search.

Since total number of BSTs with n nodes is given by $C(2n,n)/(n+1)$, which grows exponentially, brute force is hopeless.

Example: What is an optimal BST for keys A, B, C, and D with search probabilities 0.1, 0.2, 0.4, and 0.3, respectively?

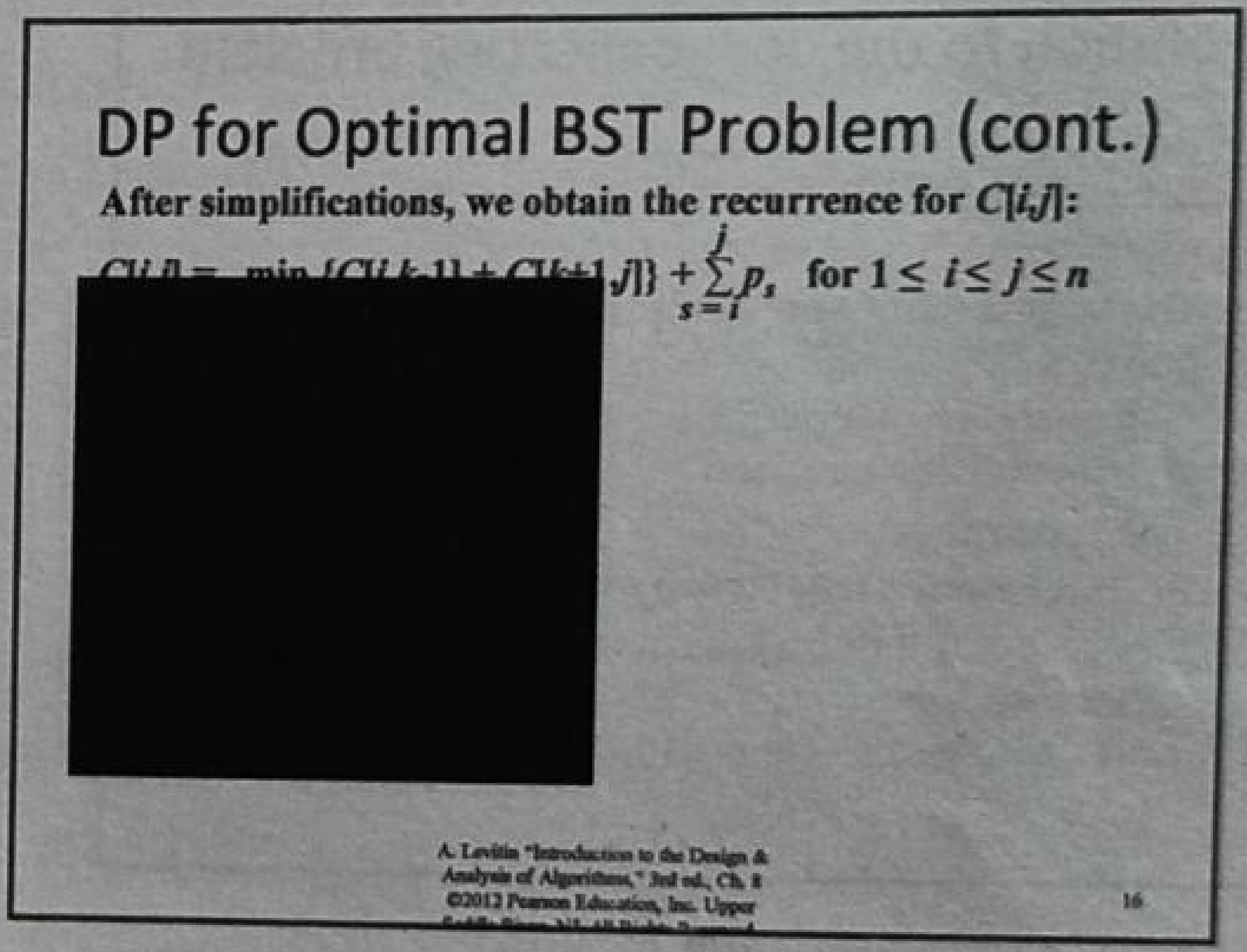


DP for Optimal BST Problem

Let $C[i,j]$ be minimum average number of comparisons made in $T[i,j]$, optimal BST for keys $a_i < \ldots < a_j$, where $1 \le i \le j \le n$. Consider optimal BST among all BSTs with some a_k ($i \le k \le j$) as their root; $T[i,j]$ is the best among them.

a_k
Optimal BST for $a_i, \ldots, a_{k-1}$
Optimal BST for $a_{k+1}, \ldots, a_j$

$$C[i,j] = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s\,(\text{level } a_s \text{ in } T[i,k-1] + 1) + \sum_{s=k+1}^{j} p_s\,(\text{level } a_s \text{ in } T[k+1,j] + 1)\}$$

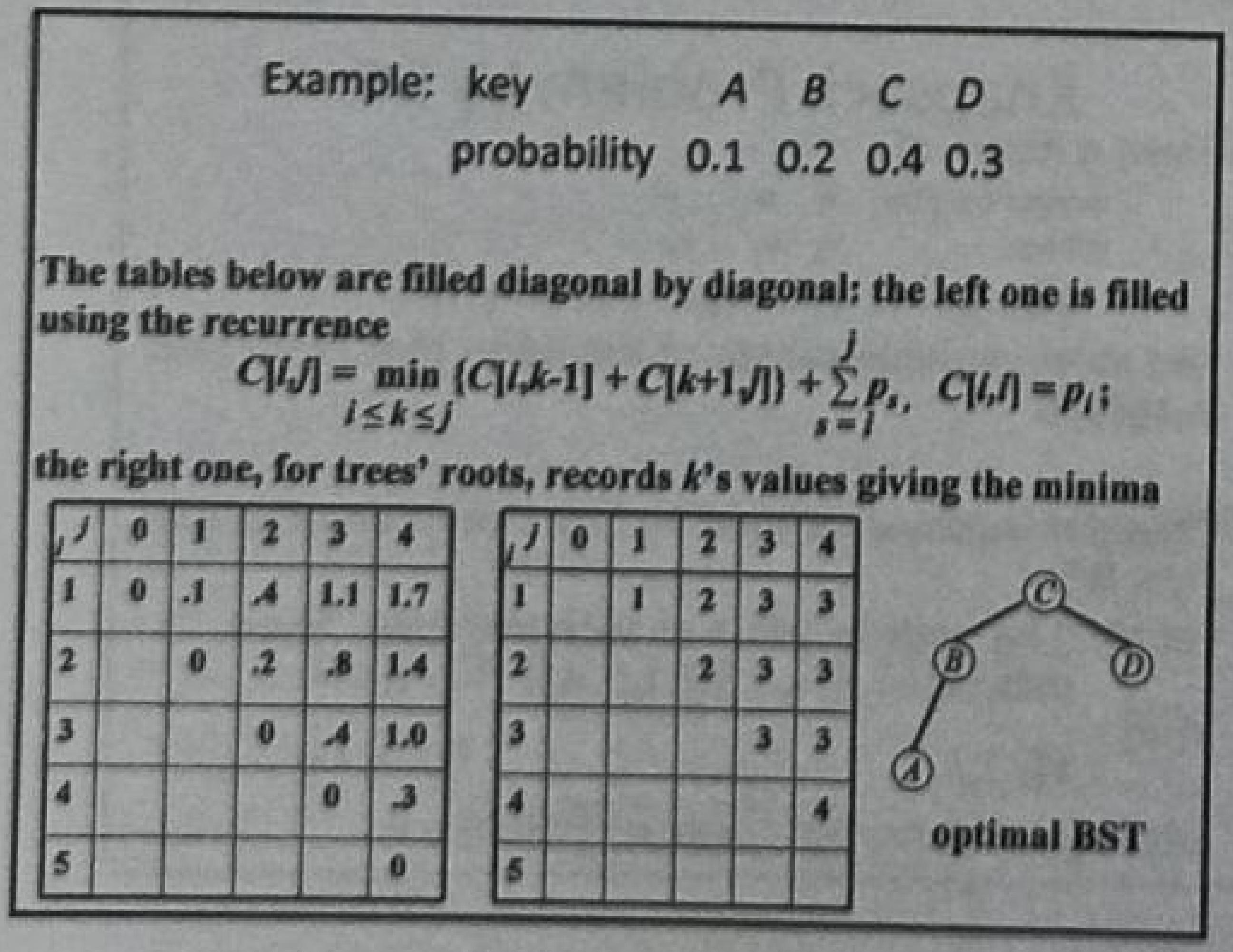


DP for Optimal BST Problem (cont.)

After simplifications, we obtain the recurrence for $C[i,j]$:

$$C[i,j] = \min_{} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s \quad \text{for } 1 \le i \le j \le n$$

[redacted]



Example: key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The tables below are filled diagonal by diagonal: the left one is filled using the recurrence

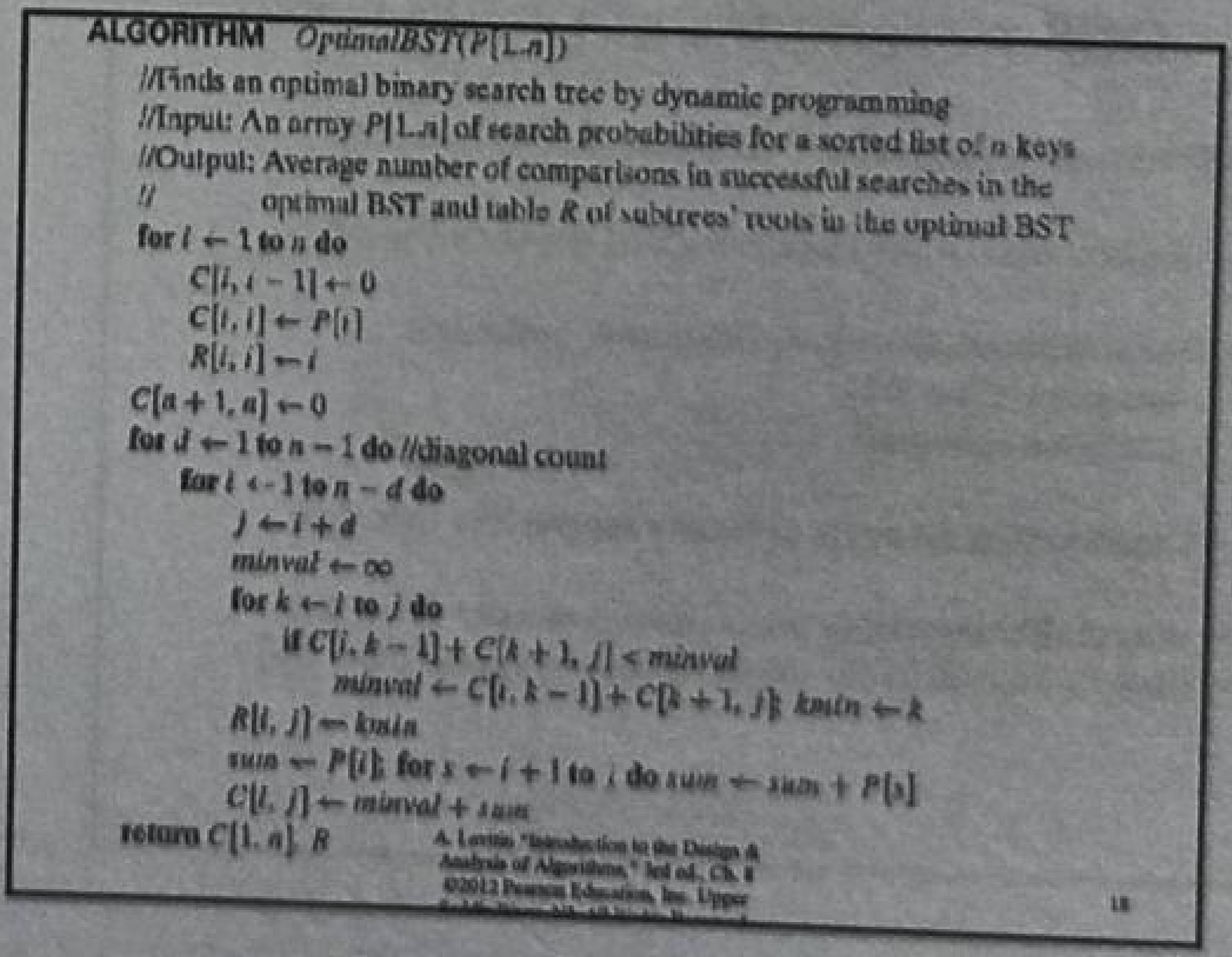
$$C[i,j] = \min_{i \le k \le j} \{C[i,k-1] + C[k+1,j]\} + \sum_{s=i}^{j} p_s, \quad C[i,i] = p_i;$$

the right one, for trees' roots, records k's values giving the minima

i \ j	0	1	2	3	4
1	0	.1	.4	1.1	1.7
2		0	.2	.8	1.4
3			0	.4	1.0
4				0	.3
5					0

i \ j	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

optimal BST



```
ALGORITHM OptimalBST(P[1..n])
    //Finds an optimal binary search tree by dynamic programming
    //Input: An array P[1..n] of search probabilities for a sorted list of n keys
    //Output: Average number of comparisons in successful searches in the
    //          optimal BST and table R of subtrees' roots in the optimal BST
    for i ← 1 to n do
        C[i, i − 1] ← 0
        C[i, i] ← P[i]
        R[i, i] ← i
    C[n + 1, n] ← 0
    for d ← 1 to n − 1 do //diagonal count
        for i ← 1 to n − d do
            j ← i + d
            minval ← ∞
            for k ← i to j do
                if C[i, k − 1] + C[k + 1, j] < minval
                    minval ← C[i, k − 1] + C[k + 1, j]; kmin ← k
            R[i, j] ← kmin
            sum ← P[i]; for s ← i + 1 to j do sum ← sum + P[s]
            C[i, j] ← minval + sum
    return C[1, n], R
```

Dynamic Programming

Dynamic Programming **is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems**

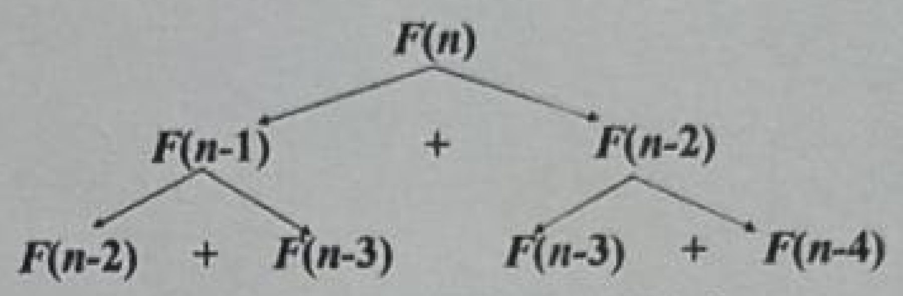
- **Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS**
- **"Programming" here means "planning"**
- **Main idea:**
  - **set up a recurrence relating a solution to a larger instance to solutions of some smaller instances**
  - **solve smaller instances once**
  - **record solutions in a table**
  - **extract solution to the initial instance from that table**

Example 1: Fibonacci numbers

- **Recall definition of Fibonacci numbers:**

$F(n) = F(n\text{-}1) + F(n\text{-}2)$
$F(0) = 0$
$F(1) = 1$

- **Computing the n^{th} Fibonacci number recursively (top-down):**



Example 1: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n\text{-}2) =$
$F(n\text{-}1) =$
$F(n) = F(n\text{-}1) + F(n\text{-}2)$

0	1	1	. . .	$F(n\text{-}2)$	$F(n\text{-}1)$	$F(n)$

Efficiency:
- time
- space

Example 2: Coin-row problem

There is a row of *n* coins whose values are some positive integers c□, c□,...,c_n, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

E.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

DP solution to the coin-row problem

Let F(*n*) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F(*n*), we partition all the allowed coin selections into two groups:

those without last coin – the max amount is ?
those with the last coin -- the max amount is ?

Thus we have the following recurrence

$F(n) = \max\{c_n + F(n\text{-}2),\ F(n\text{-}1)\}$ for $n > 1$,

$F(0) = 0,\ F(1)=c□$

DP solution to the coin-row problem (cont.)

$F(n) = \max\{c_n + F(n\text{-}2),\ F(n\text{-}1)\}$ for $n > 1$,

$F(0) = 0,\ F(1)=c□$

index	0	1	2	3	4	5	6
coins	–	5	1	2	10	6	2
F( )							

Max amount:
Coins of optimal solution:
Time efficiency:
Space efficiency:

Note: All smaller instances were solved.

Why discussion on NPC

- If a problem is proved to be NPC, a good evidence for its intractability (hardness).
- Not waste time on trying to find efficient algorithm for it
- Instead, focus on design approximate algorithm or a solution for a special case of the problem
- Some problems looks very easy on the surface, but in fact, is hard (NPC).

Decision VS. Optimization Problems

- Decision problem: solving the problem by giving an answer "YES" or "NO"
- Optimization problem: solving the problem by finding the optimal solution.
- Examples:
  - SHORTEST-PATH (optimization)
    - Given *G*, *u*,*v*, find a path from *u* to *v* with fewest edges.
  - PATH (decision)
    - Given *G*, *u*,*v*, and *k*, whether exist a path from *u* to *v* consisting of at most *k* edges.

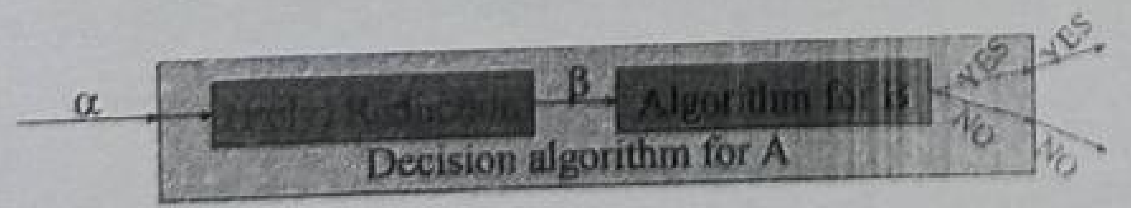
Decision VS. Optimization Problems (Cont.)

- Decision is easier (i.e., no harder) than optimization
- If there is an algorithm for an optimization problem, the algorithm can be used to solve the corresponding decision problem.
  - Example: SHORTEST-PATH for PATH
- If optimization is easy, its corresponding decision is also easy. Or in another way, if provide evidence that decision problem is hard, then the corresponding optimization problem is also hard.
- NPC is confined to decision problem. (also applicable to optimization problem.)
  - Another reason is that: easy to define reduction between decision problems.

(Poly) reduction between decision problems

- Problem (class) and problem instance
- Instance α of decision problem A and instance β of decision problem B
- A reduction from A to B is a transformation with the following properties:
  - The transformation takes poly time
  - The answer is the same (the answer for α is YES if and only if the answer for β is YES).

Implication of (poly) reduction



1. If decision algorithm for B is poly, so does A.
   A is no harder than B (or B is no easier than A)
2. If A is hard (e.g., NPC), so does B.
3. How to prove a problem B to be NPC ??
   (at first, prove B is in NP, which is generally easy.)
   3.1 find a already proved NPC problem A
   3.2 establish an (poly) reduction from A to B

Question: What is and how to prove the first NPC problem?
Circuit-satisfiability problem.

Discussion on Poly time problems

- $\Theta(n^{100})$ vs. $\Theta(2^n)$
  - Reasonable to regard a problem of $\Theta(n^{100})$ as intractable, however, very few practical problem with $\Theta(n^{100})$.
  - Most poly time algorithms require much less.
  - Once a poly time algorithm is found, more efficient algorithm may follow soon.
- Poly time keeps same in many different computation models, e.g., poly class of serial random-access machine ≡ poly class of abstract Turing machine ≡ poly class of parallel computer (#processors grows polynomially with input size)
- Poly time problems have nice closure properties under addition, multiplication and composition.

Encoding impact on complexity

- The problem instance must be represented in a way the program (or machine) can understand.
- General encoding is "binary representation".
- Different encoding will result in different complexities.
- Example: an algorithm, only input is integer k, running time is $\Theta(k)$.
  - If k is represented in *unary*: a string of k 1s, the running time is $\Theta(k) = \Theta(n)$ on length-n input, poly on n.
  - If k is represented in *binary*: the input length $n = \lfloor \log k \rfloor + 1$, the running time is $\Theta(k) = \Theta(2^n)$, exponential on n.
- Ruling out *unary*, other encoding methods are same.

13

Examples of encoding and complexity

- Given integer n, check whether n is a composite.
- Dynamic programming for subset-sum.

14

Class P Problems

- Let n= the length of binary encoding of a problem (i.e., input size), $T(n)$ is the time to solve it.
- A problem is *poly-time solvable* if $T(n) = O(n^k)$ for some constant k.
- Complexity class **P**=set of problems that are *poly-time solvable*.

15

Poly Time Verification

- PATH problem: Given $<G,u,v,k>$, whether exists a path from u to v with at most k edges?
- Moreover, also given a path p from u to v, verify whether the length of p is at most k?
- Easy or not?

Of course, very easy.

16

Poly Time Verification, encoding, and language

- Hamiltonian cycles
  - A simple path containing every vertex.
  - HAM-CYCLE={<G>: G is a Hamiltonian graph, i.e. containing Hamiltonian cycle}.
  - Suppose n is the length of **encoding** of G.
  - HAM-CYCLE can be considered as a **Language** after encoding, i.e. a subset of Σ^* where $\Sigma=\{0,1\}^*$.
- The naïve algorithm for determining HAM-CYCLE runs in $\Omega(m!)=\Omega(2^m)$ time, where m is the number of vertices, $m \approx n^{1/2}$.
- However, given an ordered sequence of m vertices (called "certificate"), let you verify whether the sequence is a Hamiltonian cycle. Very easy. In $O(n^2)$ time.

17

Class NP problems

- For a problem p, given its certificate, the certificate can be verified in poly time.
- Call this kind of problem an NP one.
- Complement set/class: Co-NP.
  - Given a set S (as a universal) and given a subset A
  - The complement is that S-A.
  - When NP problems are represented as languages (i.e. a set), we can discuss their complement set, i.e., Co-NP.

18

Relation among P, NP and co-NP=$\{L: \overline{L} \in NP$ where $\overline{L} = \Sigma^*-L\}$

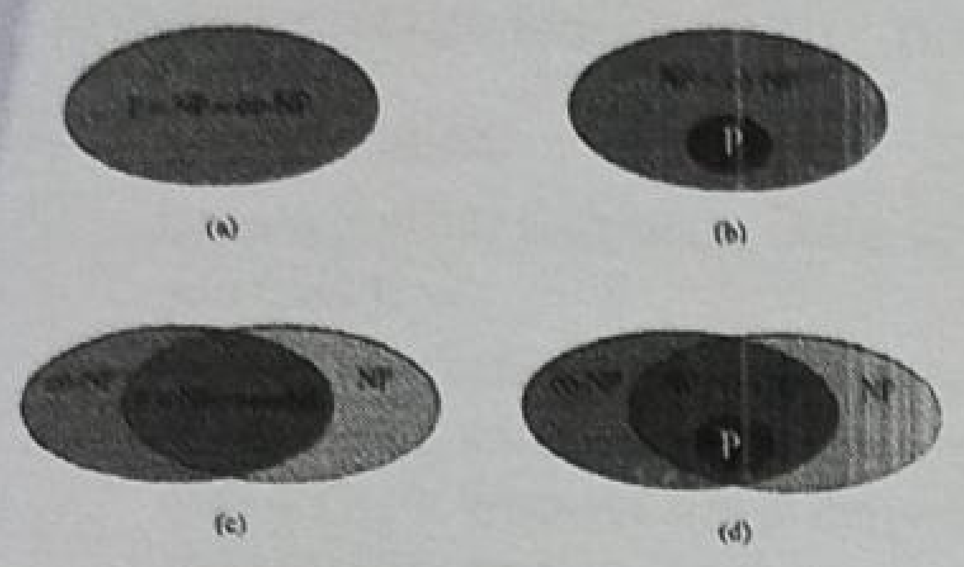


Figure 34.3 Four possibilities for relationships among complexity classes. In each diagram, one region enclosing another indicates a proper-subset relation. (a) P = NP = co-NP. Most researchers regard this possibility as the most unlikely. (b) If NP is closed under complement, then NP = co-NP, but it need not be the case that P = NP. (c) P = NP ∩ co-NP, but NP is not closed under complement. (d) NP ≠ co-NP and P ≠ NP ∩ co-NP. Most researchers regard this possibility as the most likely.

NP-completeness and Reducibility

- A (class of) problem P_1 is poly-time reducible to P_2, written as $P_1 \leq_p P_2$ if there exists a poly-time function f: $P_1 \rightarrow P_2$ such that for any instance of $p_1 \in P_1$, p_1 has "YES" answer if and only if answer to $f(p_1)$ ($\in P_2$) is also "YES".
- *Theorem 34.3*: (page 985)
  - -- For two problems P_1, P_2, if $P_1 \leq_p P_2$ then $P_2 \in P$ implies $P_1 \in P$.

NP-completeness and Reducibility (cont.)

- A problem p is NP-complete if
  1. $p \in NP$ and
  2. $p' \leq_p p$ for every $p' \in NP$.

  (if p satisfies 2, then p is said NP-hard.)

Theorem 34.4 (page 986)

if any NP-compete problem is poly-time solvable, then P=NP. Or say: if any problem in NP is not poly-time solvable, then no NP-complete problem is poly-time solvable.

First NP-complete problem—Circuit Satisfiability (problem definition)

- Boolean combinational circuit
  - Boolean combinational elements, wired together
  - Each element, inputs and outputs (binary)
  - Limit the number of outputs to 1.
  - Called *logic gates*: NOT gate, AND gate, OR gate.
  - *true table*: giving the outputs for each setting of inputs
  - true assignment: a set of boolean inputs.
  - satisfying assignment: a true assignment causing the output to be 1.
  - A circuit is satisfiable if it has a satisfying assignment.

Circuit Satisfiability Problem: definition

- Circuit satisfying problem: given a boolean combinational circuit composed of AND, OR, and NOT, is it stisfiable?
- CIRCUIT-SAT={<C>: C is a satisfiable boolean circuit}
- Implication: in the area of computer-aided hardware optimization, if a subcircuit always produces 0, then the subcircuit can be replaced by a simpler subcircuit that omits all gates and just output a 0.

Two instances of circuit satisfiability problems

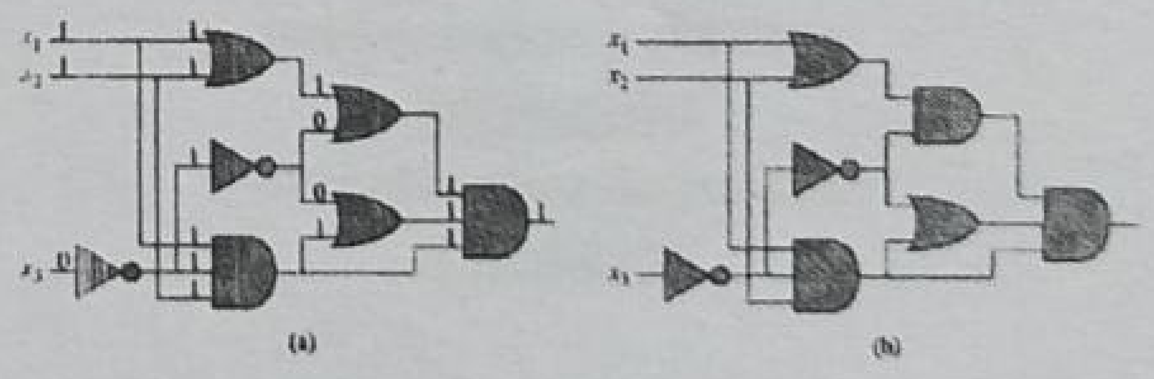


Figure 34.8 Two instances of the circuit-satisfiability problem. (a) The assignment $\langle x_1 = 1, x_2 = 1, x_3 = 0\rangle$ to the inputs of this circuit causes the output of the circuit to be 1. The circuit is therefore satisfiable. (b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. The circuit is therefore unsatisfiable.

Circuit-satisfiability problem is NP-hard (cont.)

- The reduction algorithm F constructs a single combinational circuit C as follows:
  - Paste together all $T(n)$ copies of the circuit M.
  - The output of the ith circuit, which produces c_i, is directly fed into the input of the $(i+1)$st circuit.
  - All items in the initial configuration, except the bits corresponding to certificate y, are wired directly to their known values.
  - The bits corresponding to y are the inputs to C.
  - All the outputs to the circuit are ignored, except the one bit of $c_{T(n)}$ corresponding to the output of A.

31

Circuit-satisfiability problem is NP-hard (cont.)

- Two properties remain to be proven:
  - F correctly constructs the reduction, i.e., C is satisfiable if and only if there exists a certificate y, such that $A(x,y)=1$.
    - $\Leftarrow$Suppose there is a certificate y, such that $A(x,y)=1$. Then if we apply the bits of y to the inputs of C, the output of C is the bit of $A(x,y)$, that is $C(y)=A(x,y)=1$, so C is satisfiable.
    - $\Rightarrow$Suppose C is satisfiable, then there is a y such that $C(y)=1$. So, $A(x,y)=1$.
  - F runs in poly time.

32

Circuit-satisfiability problem is NP-hard (cont.)

- F runs in poly time.
  - Poly space
    - Size of x is n.
    - Size of A is constant, independent of x.
    - Size of y is $O(n^k)$.
    - Amount of working storage is poly in n since A runs at most $O(n^k)$.
    - M has size poly in length of configuration, which is poly in $O(n^k)$, and hence is poly in n.
    - C consists of at most $O(n^k)$ copies of M, and hence is poly in n.
    - Thus, the C has poly space.
  - The construction of C takes at most $O(n^k)$ steps and each step takes poly time, so F takes poly time to construct C from x.

33

CIRCUIT-SAT is NP-complete

- In summary
  - CIRCUIT-SAT belongs to NP, verifiable in poly time.
  - CIRCUIT-SAT is NP-hard, every NP problem can be reduced to CIRCUIT-SAT in poly time.
  - Thus CIRCUIT-SAT is NP-complete.

34

NP-completeness proof basis

- *Lemma 34.8* (page 995)
  - If X is a problem (class) such that $P' \leq_p X$ for some $P' \in$ NPC, then X is NP-hard. Moreover, if $X \in$ NP, then $X \in$ NPC.
- Steps to prove X is NP-complete
  - Prove $X \in$ NP.
    - Given a certificate, the certificate can be verified in poly time.
  - Prove X is NP-hard.
    - Select a known NP-complete P'.
    - Describe a transformation function f that maps every instance x of P' into an instance $f(x)$ of X.
    - Prove f satisfies that the answer to $x \in P'$ is YES if and only if the answer to $f(x) \in X$ is YES for all instance $x \in P'$.
    - Prove that the algorithm computing f runs in poly-time.

35

NPC proof –Formula Satisfiability (SAT)

- SAT definition
  - n boolean variables: $x_1, x_2, \ldots, x_n$.
  - M boolean connectives: any boolean function with one or two inputs and one output, such as $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \ldots$ and
  - Parentheses.
- A SAT ϕ is satisfiable if there exists an true assignment which causes ϕ to evaluate to 1.
- SAT={< ϕ>: ϕ is a satifiable boolean formula}.
- The historical honor of the first NP-complete problem ever shown.

36

SAT is NP-complete

- *Theorem 34.9:* (page 997)
  - SAT is NP-complete.
- Proof:
  - SAT belongs to NP.
    - Given a satisfying assignment, the verifying algorithm replaces each variable with its value and evaluates the formula, in poly time.
  - SAT is NP-hard (show CIRCUIT-SAT$\leq_p$ SAT).

37

SAT is NP-complete (cont.)

- CIRCUIT-SAT$\leq_p$ SAT, i.e., any instance of circuit satisfiability can be reduced in poly time to an instance of formula satisfiability.
- Intuitive induction:
  - Look at the gate that produces the circuit output.
  - Inductively express each of gate's inputs as formulas.
  - Formula for the circuit is then obtained by writing an expression that applies the gate's function to its input formulas.
- Unfortunately, this is not a poly reduction
  - Shared formula (the gate whose output is fed to 2 or more inputs of other gates) cause the size of generated formula to grow exponentially

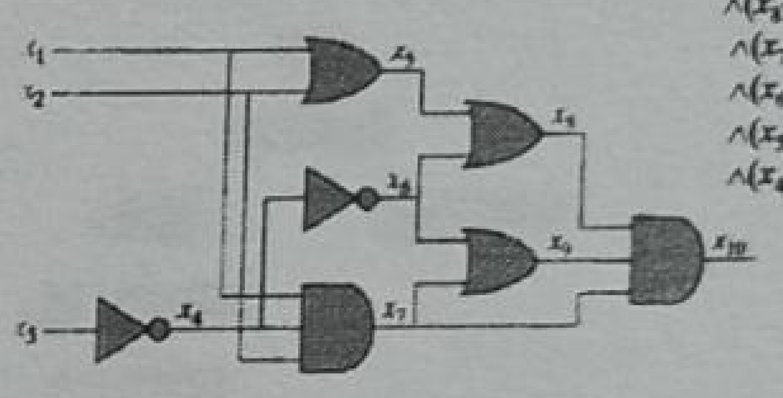
38

SAT is NP-complete (cont.)

- Correct reduction:
  - For every wire x_i of C, give a variable x_i in the formula.
  - Every gate can be expressed as $x_o \leftrightarrow (x_{i1} \theta x_{i2} \theta \ldots \theta x_{il})$
  - The final formula ϕ is the AND of the circuit output variable and conjunction of all clauses describing the operation of each gate. (example Figure 34.10)
- Correctness of the reduction
  - Clearly the reduction can be done in poly time.
  - C is satisfiable if and only if ϕ is satisfiable.
    - If C is satisfiable, then there is a satisfying assignment. This means that each wire of C has a well-defined value and the output of C is 1. Thus the assignment of wire values to variables in ϕ makes each clause in ϕ evaluate to 1. So ϕ is 1.
    - The reverse proof can be done in the same way.

39

Example of reduction of CIRCUIT-SAT to SAT



$$\begin{aligned}\phi = x_{10} &\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9))\\ &\wedge (x_9 \leftrightarrow (x_6 \vee x_7))\\ &\wedge (x_8 \leftrightarrow (x_5 \vee x_6))\\ &\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))\\ &\wedge (x_6 \leftrightarrow \neg x_4))\\ &\wedge (x_5 \leftrightarrow (x_1 \vee x_2))\\ &\wedge (x_4 \leftrightarrow \neg x_3)\end{aligned}$$

Figure 34.10 Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

INCORRECT REDUCTION: $\phi = x_{10} = x_7 \wedge x_8 \wedge x_9 = (x_1 \wedge x_2 \wedge x_4) \wedge (x_5 \vee x_6) \wedge (x_6 \vee x_7)$
$= (x_1 \wedge x_2 \wedge x_4) \wedge ((x_1 \vee x_2) \vee \neg x_4) \wedge (\neg x_4 \vee (x_1 \wedge x_2 \wedge x_4)) = \ldots$

40

NPC Proof –3-CNF Satisfiability

- 3-CNF definition
  - A *literal* in a boolean formula is an occurrence of a variable or its negation.
  - CNF (Conjunctive Nornal Form) is a boolean formula expressed as AND of clauses, each of which is the OR of one or more literals.
  - 3-CNF is a CNF in which each clause has exactly 3 distinct literals (a literal and its negation are distinct)
- 3-CNF-SAT: whether a given 3-CNF is satiafiable?

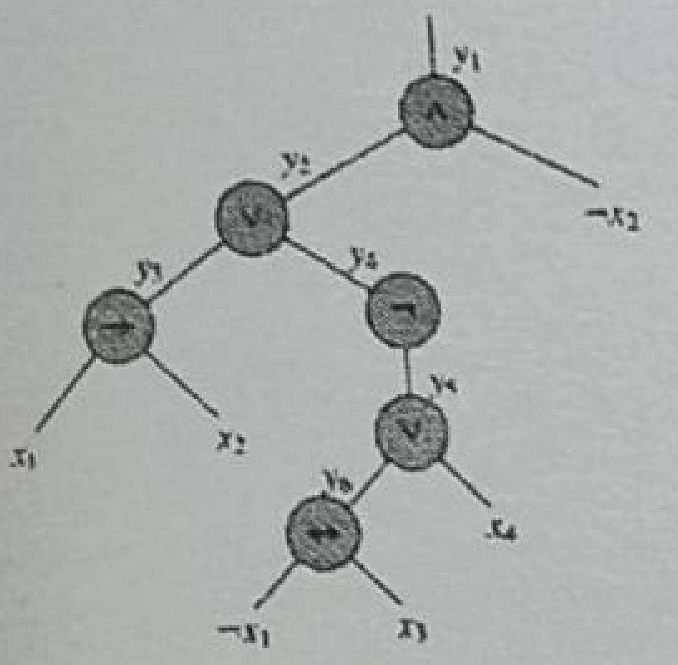
41

3-CNF-SAT is NP-complete

- Proof: 3-CNF-SAT $\in$NP. Easy.
  - 3-CNF-SAT is NP-hard. (show SAT $\leq_p$3-CNF-SAT)
    - Suppose ϕ is any boolean formula, Construct a binary 'parse' tree, with literals as leaves and connectives as internal nodes.
    - Introduce a variable y_i for the output of each internal nodes.
    - Rewrite the formula to ϕ' as the AND of the root variable and a conjunction of clauses describing the operation of each node.
    - The result is that in ϕ', each clause has at most three literals.
    - Change each clause into conjunctive normal form as follows:
      - Construct a true table, (small, at most 8 by 4)
      - Write the disjunctive normal form for all true-table items evaluating to 0
      - Using DeMorgan law to change to CNF.
    - The resulting ϕ'' is in CNF but each clause has 3 or less literals.
    - Change 1 or 2-literal clause into 3-literal clause as follows:
      - If a clause has one literal l, change it to $(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$.
      - If a clause has two literals $(l_1 \vee l_2)$, change it to $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$.

42

Binary parse tree for $\phi=((x_1\to x_2)\vee\neg((\neg x_1\leftrightarrow x_3)\vee x_4))\wedge\neg x_2$



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

Figure 34.11 The tree corresponding to the formula $\phi=((x_1\to x_2)\vee\neg((\neg x_1\leftrightarrow x_3)\vee x_4))\wedge\neg x_2$

Example of Converting a 3-literal clause to CNF format

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

Disjunctive Normal Form

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

Conjunctive Normal Form

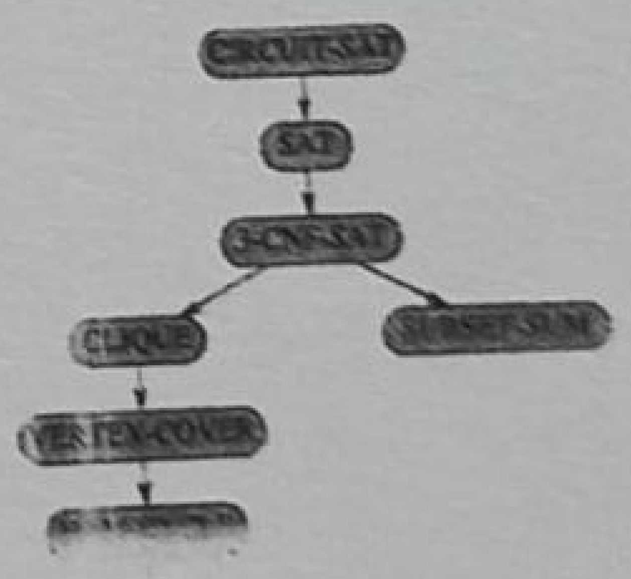
$$\phi_i''=(\neg y_1\vee\neg y_2\vee\neg x_2)\wedge(\neg y_1\vee y_2\vee\neg x_2)\wedge(\neg y_1\vee y_2\vee x_2)\wedge(y_1\vee\neg y_2\vee x_2)$$

Figure 34.12 The truth table for the clause $(y_1\leftrightarrow(y_2\wedge\neg x_2))$.

3-CNF is NP-complete

- ϕ and reduced 3-CNF are equivalent:
  - From ϕ to ϕ', keep equivalence.
  - From ϕ' to ϕ'', keep equivalence.
  - From ϕ'' to final 3-CNF, keep equivalence.
- Reduction is in poly time,
  - From ϕ to ϕ', introduce at most 1 [illegible]
  - [illegible]
  - From ϕ'' to final 3-CNF, [illegible] clause in ϕ''.

NP-completeness proof structure



[illegible]