Analysis of Algorithms

Review

What is an Algorithm ?

An *algorithm* is a finite, clearly specified sequence of instructions to be followed to solve a problem or compute a function

An *algorithm* generally

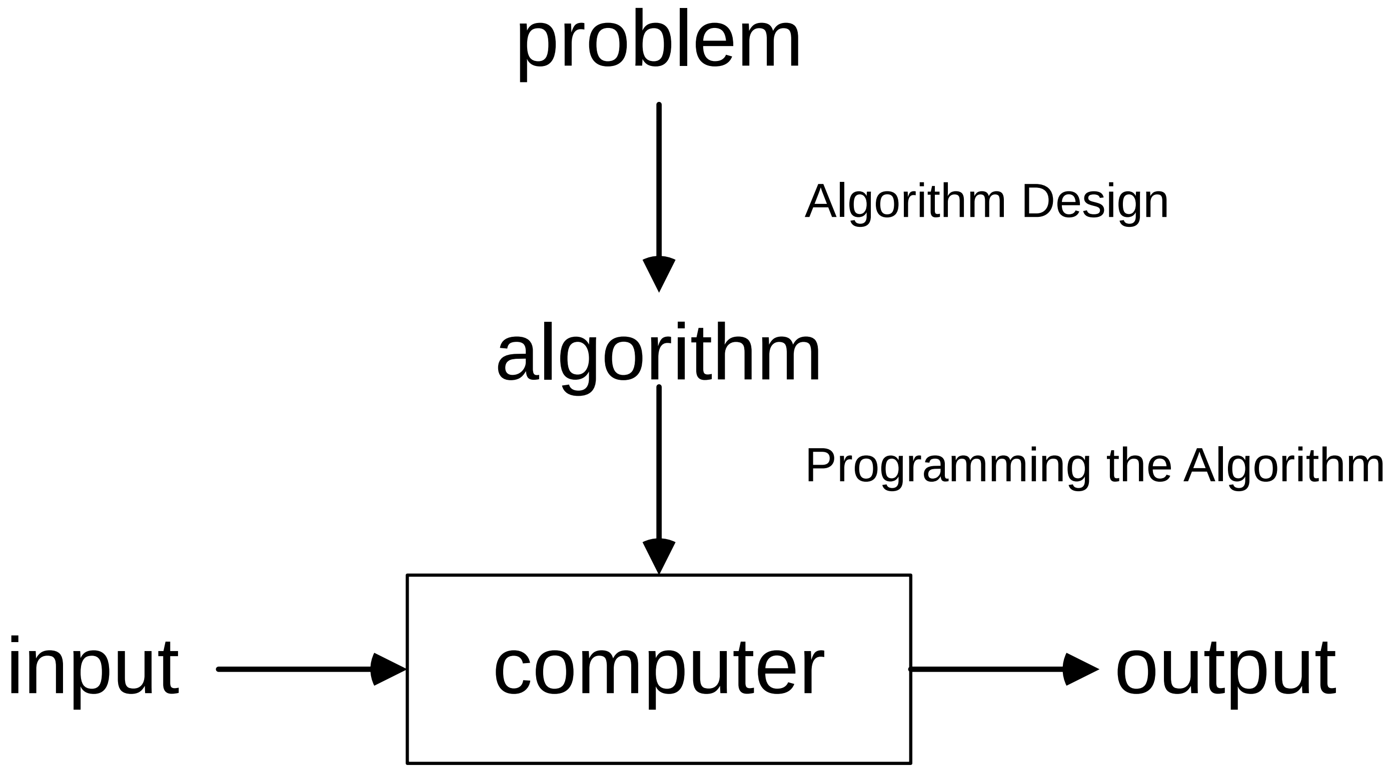
- takes some input
- carries out a number of effective instructions in a finite amount of time
- produces some output.

An effective instruction is an operation so basic that it is possible to carry it out using pen and paper.

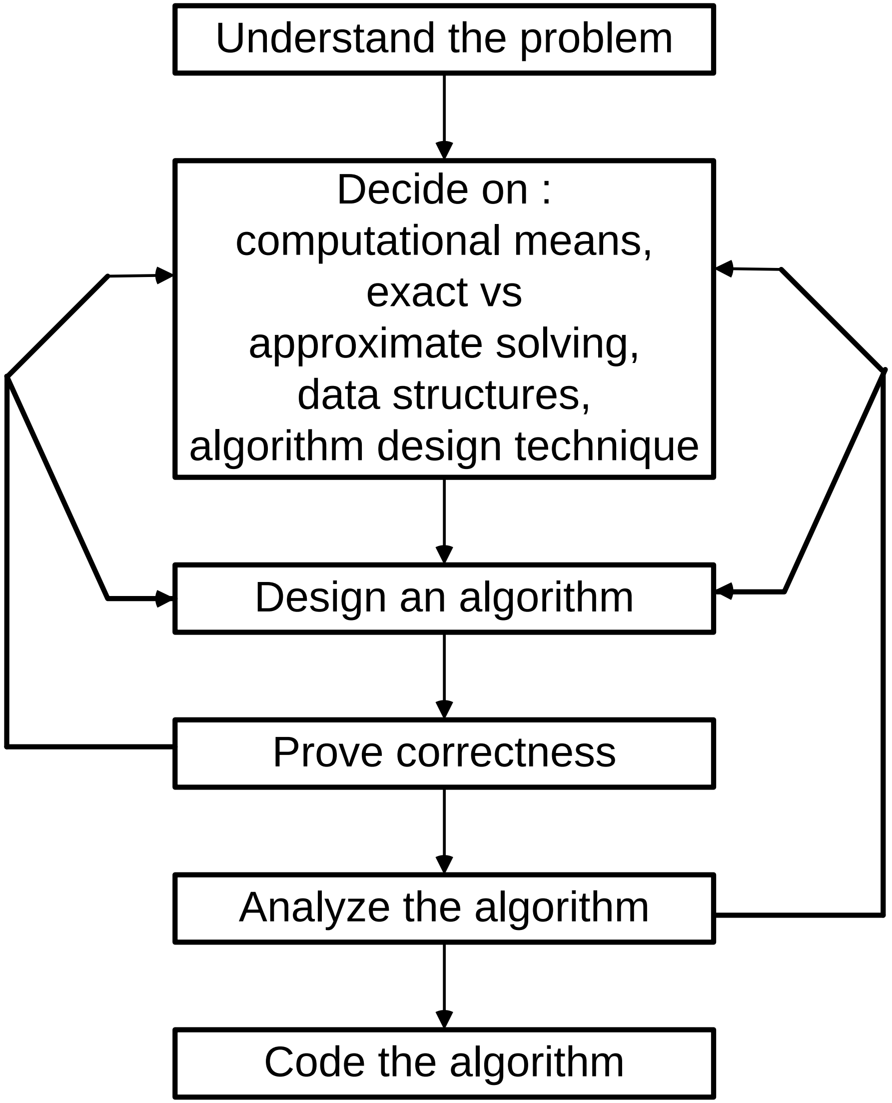
Properties of an Algorithm

- **Effectiveness**
  - Instructions are simple
    - can be carried out by pen and paper
- **Definiteness**
  - Instructions are clear
    - meaning is unique
- **Correctness**
  - Algorithm gives the right answer
    - for all possible cases
- **Finiteness**
  - Algorithm stops in reasonable time
    - produces an output

Notion of an Algorithm



Algorithm Design Process



Algorithm design techniques/strategies

- Brute force
- Decrease and conquer
- Divide and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Backtracking
- Branch and bound

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Mathematical Background

- Functions
- Logarithm
- Summation
- Probability
- Asymptotic Notations
- Recursion
  - Recurrence equation

Asymptotic notations

O notation

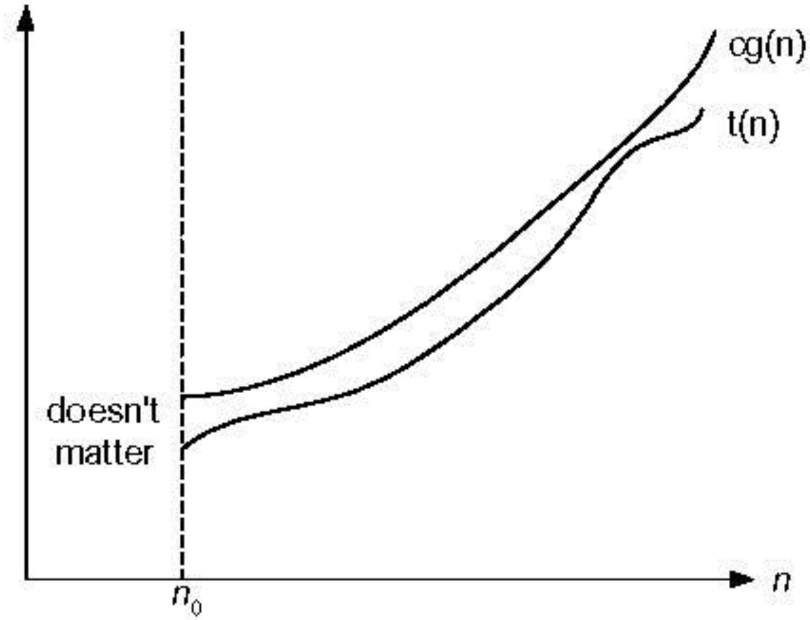


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Asymptotic notations

Ω notation

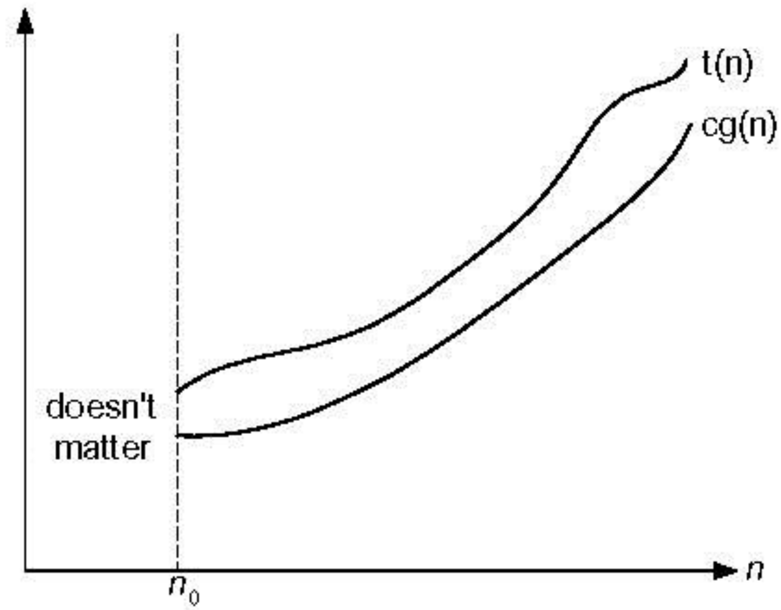


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Asymptotic notations

θ notation

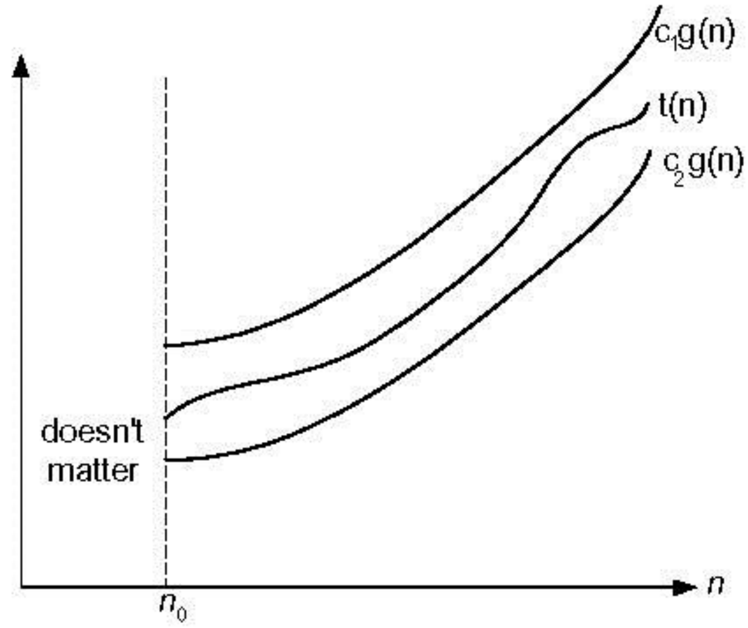


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Recurrence Relations

- **Solution Methods**
  - **Exact**
    - Forward substitution
    - Backward substitution
  - **Asymptotic**
    - Master theorem

Master Theorem

Let T(n) be an eventually nondecreasing function that satisfies the recurrence

$$T(n) = a\,T(n/b) + f(n) \quad \text{for } n = b^k, k = 1, 2, \ldots$$
$$T(1) = c,$$
$$where\ a \geq 1,\ b \geq 2, c > 0.$$
$$If\ f(n) \in \Theta\left(n^d\right)\ where\ d \geq 0, then$$

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d, \\ \Theta(n^d \log n) & \text{if } a = b^d, \\ \Theta(n^{\log_b a}) & \text{if } a > b^d. \end{cases}$$

Common Recurrence Types in Algorithm Analysis

- Decrease-by-One

$$T(n) = T(n-1) + f(n)$$

- Decrease-by-a-Constant-Factor

$$T(n) = T\left({}^{n}/_{b}\right) + f(n)$$

- Divide-and-Conquer

$$T(n) = a\,T\left({}^{n}/_{b}\right) + f(n)$$

Brute Force and Exhaustive Search

- Selection Sort
- Bubble Sort
- Sequential Search
- String Matching
- Travelling Salesman Problem
- Knapsack Problem
- Assignment Problem
- Depth-First Search
- Breadth-First Search

Brute Force

- Applicable to wide variety of problems
- Easiest way solving problem
  - Do not think much
  - Just do it!..
- ***Brute force*** is a straightforward approach based on
  - the problem's statement
  - definitions of the concepts

Decrease And Conquer

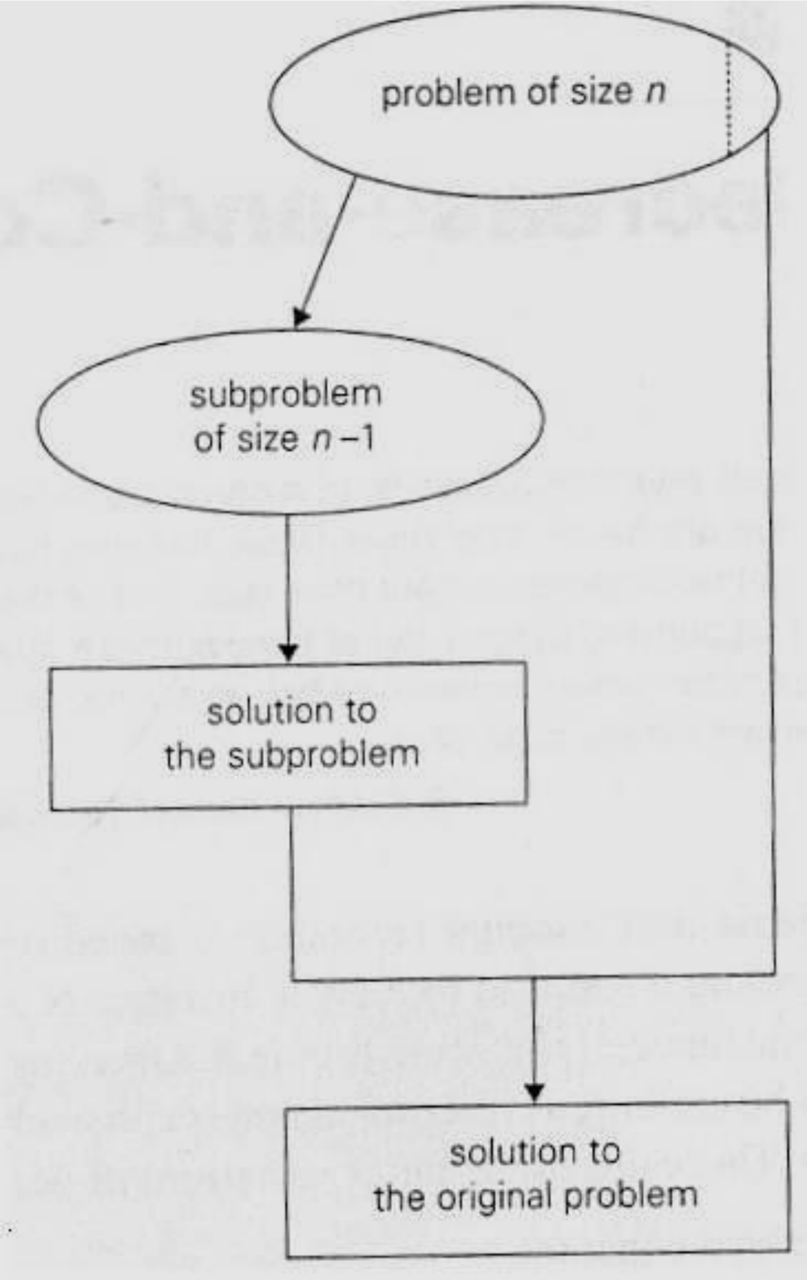
Decrease and conquer technique is based on

- To solve a *problem,* use a solution of a *smaller instance of the same problem*
- it can be done either top down (recursively) or bottom up (without a recursion)
- Variations of decrease and conquer :
  1. Decrease by a constant
  2. Decrease by a constant factor
  3. Variable size decrease

Decrease And Conquer

1. Decrease by a constant

- Size of an instance is reduced by the same constant on each iteration of the algorithm
  - typically this constant is equal to one



Decrease (by one) and conquer technique

Decrease And Conquer

Example :Computing a^n for a positive integer a.

- Relationship between a solution to an instance of size *n* and size *n-1* is obtained by formula

$$a^n = a^{n-1}.a \qquad\qquad f(n) = a^n$$

- can be computed topdown by using recursive definition

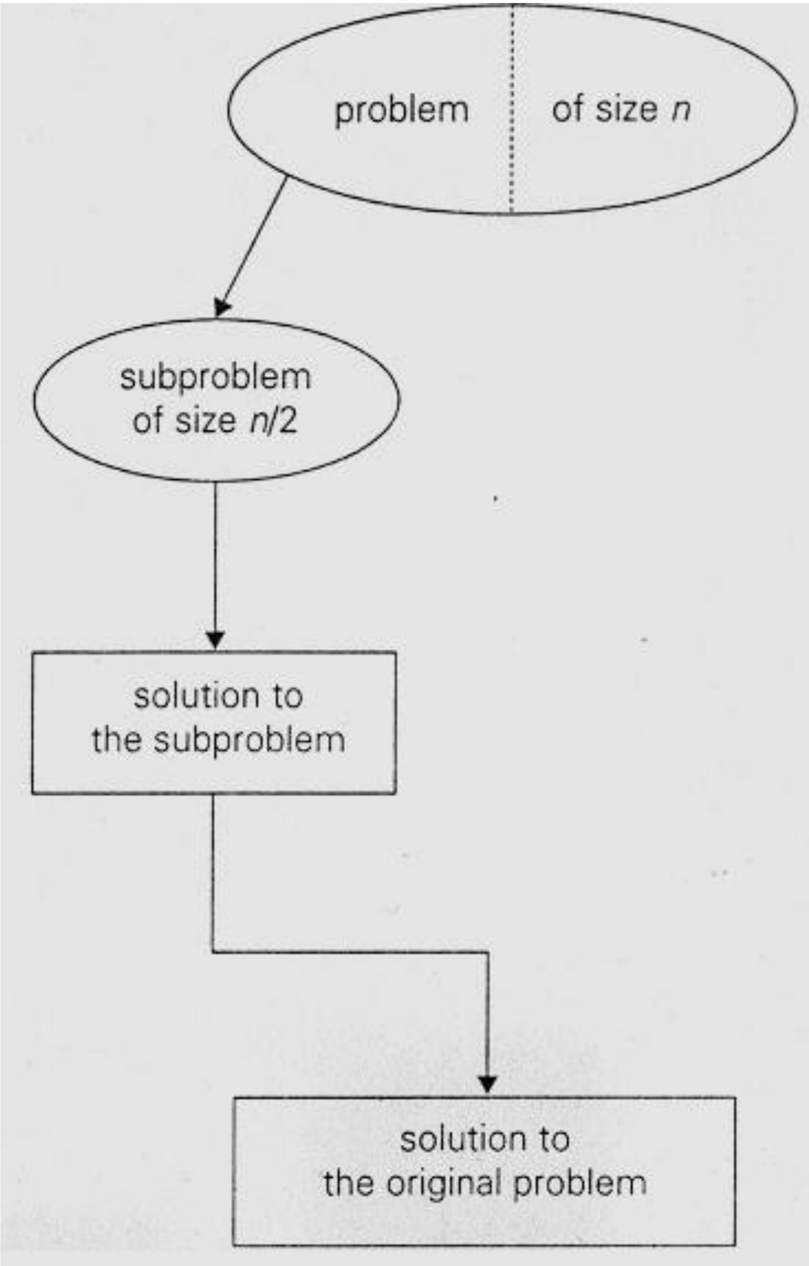
$$f(n) = \begin{cases} f(n-1).a & if \;\; \mathrm{n} > 1 \\ a & if \;\; n = 1 \end{cases}$$

- can be computed bottom up by multiplying *a* by *n-1* times
    - same as brute force

Decrease And Conquer

2. Decrease by a constant factor

- Reduce a problem’s instance by the some constant factor on each iteration of the algorithm
  - in most applications this constant is two



Decrease (by half) and conquer technique

Decrease And Conquer

Example: Computing a^n for positive integer a

- If the instance of size *n* is to compute a^n, the instance of half its size will be to compute $a^{n/2}$

$$a^n = \left(a^{n/2}\right)^2$$

Does this work for all integers ?

Decrease And Conquer

The formula is different for odd or even integers

$$a^n = \begin{cases} \left(a^{n/2}\right)^2 & \text{if n is even and positive} \\ \left(a^{(n-1)/2}\right)^2 . a & \text{if n is odd and greater than 1} \\ a & \text{if n} = 1 \end{cases}$$

The runtime of the algorithm is *O(logn)*

Why??

Decrease And Conquer

3. Variable size decrease

- A size reduction varies from one iteration of an algorithm to another
- EX: Euclid's algorithm for computing the greatest common divisor

`gcd (m,n) = gcd (n, m mod n)`

Arguments on right-hand side are always smaller than those on the left-hand side

- **At least starting with the second iteration of the algorithm**

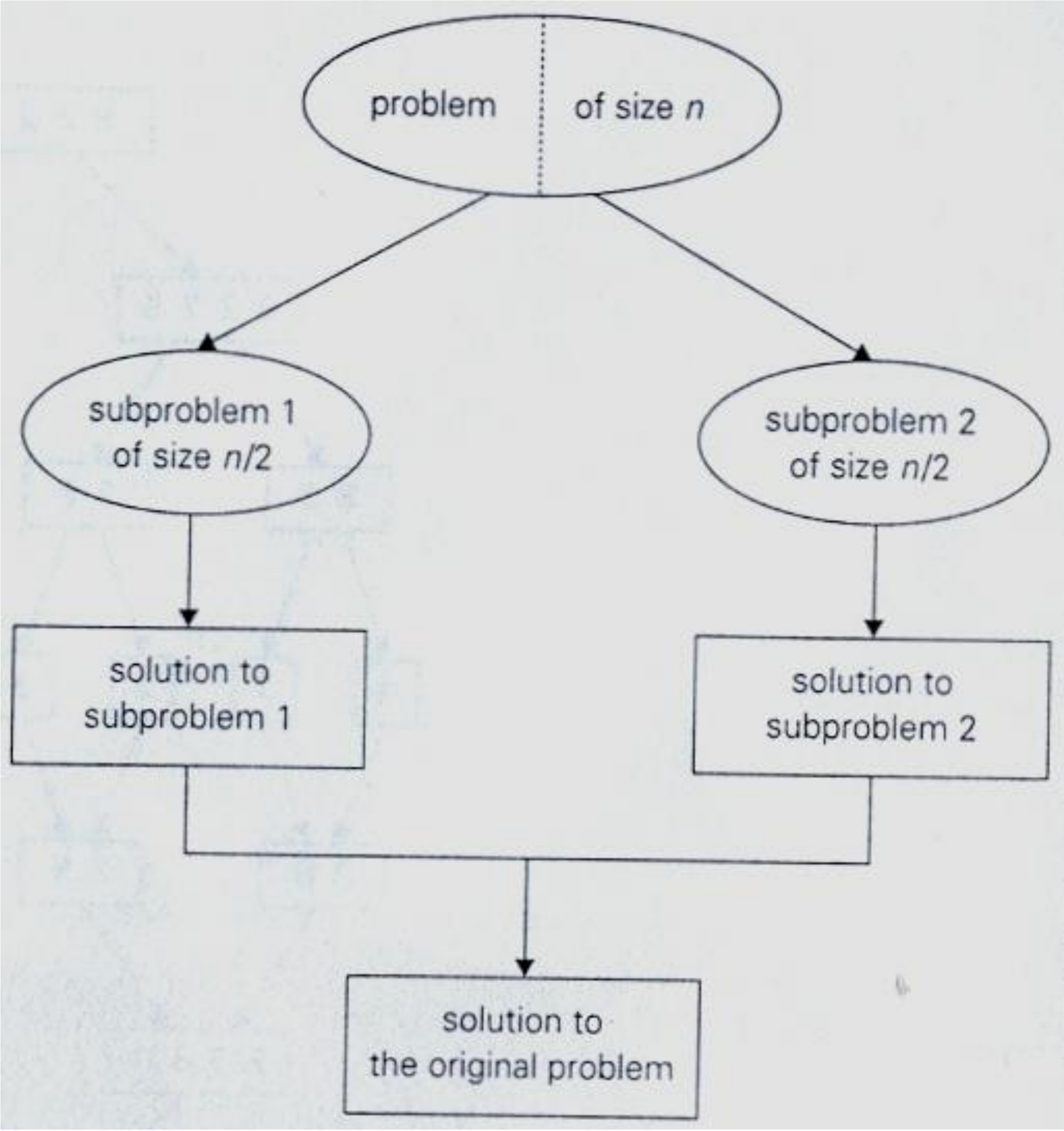
Divide And Conquer

A well known general algorithm design technique

Approach*:*

- A problem's instance is divided into several <u>smaller</u> instances of the <u>same</u> problem
  - ideally of about the same size
- The smaller instances are solved
  - typically recursively
- The solutions obtained for the smaller instances are combined to get a solution to the original problem

Divide And Conquer



General Divide-and-Conquer Recurrence

$T(n) = aT(n/b) + f(n)$ where $f(n) \in \Theta(n^d)$, $d \geq 0$

Master Theorem: If $a < b^d$, $T(n) \in \Theta(n^d)$

If $a = b^d$, $T(n) \in \Theta(n^d \log n)$

If $a > b^d$, $T(n) \in \Theta(n^{\log_b a})$

Note: The same results hold with O instead of Θ.

Examples: $T(n) = 4T(n/2) + n \Rightarrow T(n) \in$?

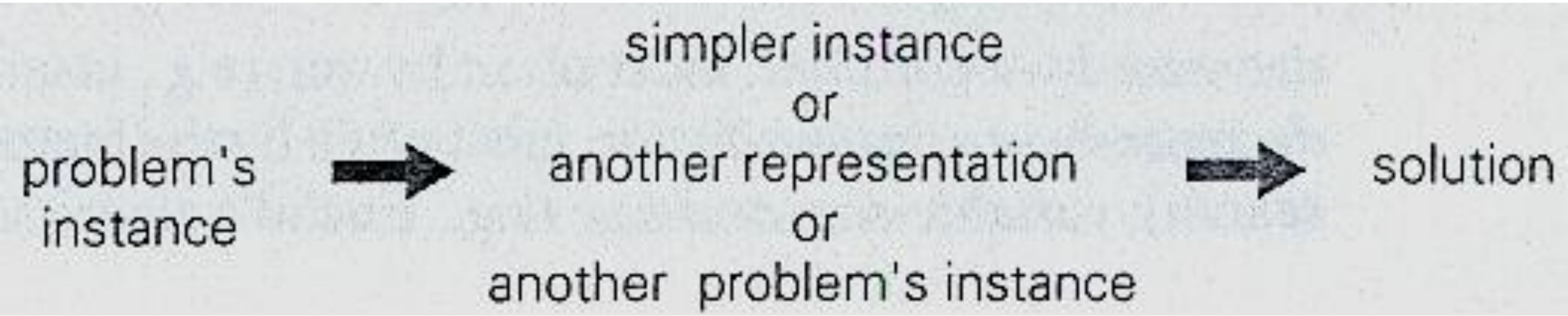
$T(n) = 4T(n/2) + n^2 \Rightarrow T(n) \in$?

$T(n) = 4T(n/2) + n^3 \Rightarrow T(n) \in$?

Transform And Conquer

- *Transform and conquer technique* is based on idea of *transformation*
- This method works in two stages
  - Transformation stage
    - The problem is modified to another problem
      - more amenable to solution
  - Conquering stage
    - It is solved

Transform And Conquer Strategy



- Instance simplification
  - Transformation to a simplier instance problem
- Representation change
  - Transformation to a different representation of <u>same</u> instance
- Problem reduction
  - Tranformation to an instance of a different problem for which an algorithm is already available

Space-for-time tradeoffs

Two varieties of space-for-time algorithms:

- *input enhancement* — preprocess the input (or its part) to store some info to be used later in solving the problem
  - counting sorts
  - string searching algorithms
- *prestructuring* — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)

Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
- always makes a shift based on the text's character *c* aligned with the <u>last</u> character in the pattern according to the shift table's entry for *c*

Shift table

- Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \qquad \textbf{(7.1)}$$

  by scanning pattern before search begins and stored in a table called *shift table*

- Shift table is indexed by text and pattern alphabet
  Eg, for `BARBER`:

character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Example of Horspool's alg. application

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R           B A R B E R
        B A R B E R       B A R B E R
          B A R B E R           B A R B E R
```

Boyer-Moore algorithm

Based on same two ideas:

- comparing pattern characters to text from right to left
- precomputing shift sizes in two tables
  - *bad-symbol table* indicates how much to shift based on text's character causing a mismatch
  - *good-suffix table* indicates how much to shift based on matched part (suffix) of the pattern

Example of Boyer-Moore alg. application

c	A	B	C	D	. . .	O	. . .	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

B E S S _ K N E W _ A B O U T _ B A O B A B

B A O B A B

$d_1 = t_1(K) - 0 = 6$ B A O B A B

$d_1 = t_1(\_) - 2 = 4$ B A O B A B

$d_2 = 5$ $d_1 = t_1(\_) - 1 = 5$

$d = \max\{4, 5\} = 5$ $d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

k	**pattern**	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

Dynamic Programming

- **Definition :**
  - Dynamic programming is an interesting algorithm design technique for *optimizing multistage decision problems*
  - Programming in the name of this technique stands for *planning*
    - Does not refer to computer programming
  - It is a technique for solving problems with overlapping subproblems
    - Typically these subproblems arise from a recurrence relations
    - Suggests solving each of the smaller subproblems only once and recording the results in a table

Dynamic Programming

- 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

Dynamic programming usually used for optimization problems

How do we get the recurrence relation?

Dynamic Programming

Principle of Optimality

A general principle that underlines dynamic programming for optimization problems.

An optimal solution to any instance of an optimization problem is composed of optimal solutions to its subinstances.

Coin-row problem

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1$$
$$F(0) = 0, \qquad F(1) = c_1.$$

ALGORITHM $CoinRow(C[1..n])$

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0$; $F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow \max(C[i] + F[i-2],\ F[i-1])$
return $F[n]$

Change-making problem

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

ALGORITHM *ChangeMaking*$(D[1..m], n)$

//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
//given amount n
//Input: Positive integer n and array $D[1..m]$ of increasing positive
//　　　integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n
$F[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
　　$temp \leftarrow \infty$; $j \leftarrow 1$
　　while $j \leq m$ **and** $i \geq D[j]$ **do**
　　　　$temp \leftarrow \min(F[i - D[j]], temp)$
　　　　$j \leftarrow j + 1$
　　$F[i] \leftarrow temp + 1$
return $F[n]$

Coin-collecting problem

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

$$F(0, j) = 0 \ \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \ \text{ for } 1 \le i \le n,$$

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

ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)
$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
    $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
    for $j \leftarrow 2$ **to** m **do**
        $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$

Knapsack Problem

- The recurrence

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

- Initial conditions

$$V[0,j] = 0 \quad \text{for} \quad j \geq 0$$

$$V[i,0] = 0 \quad \text{for} \quad i \geq 0$$

Greedy Technique

- Used for solving *optimization problems*
  - such as engineering problems
- Construct a solution through a sequence of *decision steps*
  - Each expanding a partially constructed solution
  - Until a complete solution is reached
- Similar to dynamic programming
  - but, not all possible solutions are explored

Applications of the Greedy Strategy

- Optimal solutions:
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- Approximations:
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Fractional Knapsack Problem

- ***Given :***

  w_i : weight of object *i*

  m : capacity of knapsack

  p_i : profit of all of *i* is taken

- ***Find:***

  x_i : fraction of *i* taken

- ***Feasibility:***

$$\sum_{i=1}^{n} x_i w_i \leq m$$

- ***Optimality:***

$$\text{maximize} \sum_{i=1}^{n} x_i p_i$$

Knapsack Problem

```
Algorithm Knapsack (m,n)
   for i = 1 to n
        x(i) = 0
   for i = 1 to n
        select the object (j) with largest unit value
        if (w[j] < m)
              x[j] = 1.0
              m = m – w[j]
        else
              x[j] = m/w[j]
              break
```

- <u>Example :</u>
  M = 20 n = 3
  p = (25, 24, 15) w = (18, 15, 10)

Huffman Codes

- <u>Given:</u> The characters and their frequencies
- <u>Find:</u> The coding tree
- <u>Cost :</u> Minimize the cost

$$Cost = \sum_{c \in C} f(c) \times d(c)$$

- *f(c) : frequency of c*
- *d(c) : depth of c*

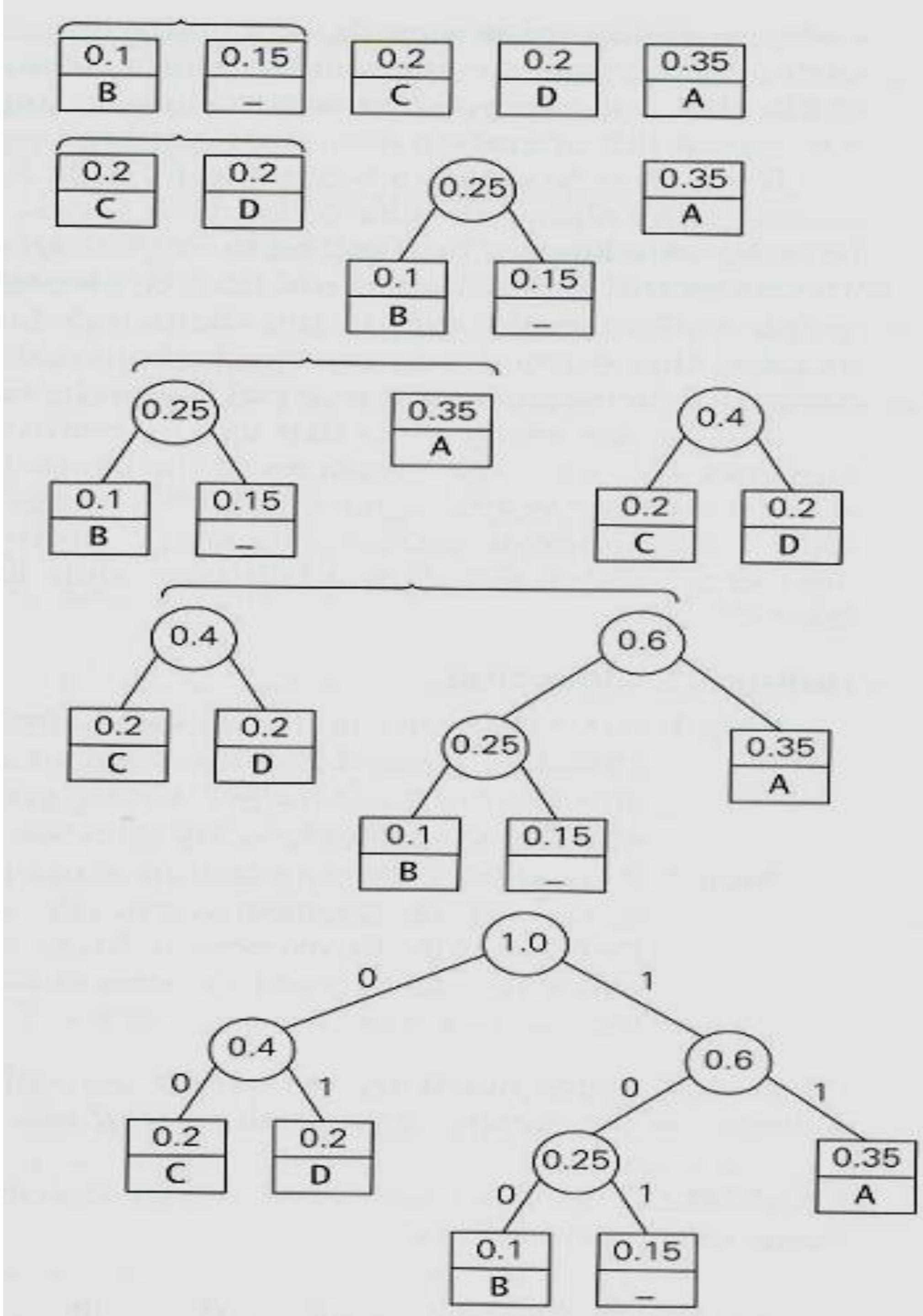
Huffman Codes Example

Consider five characters {A,B,C,D,-} with following occurrence probabilities

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is as follows

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101


0.1 B
0.15 _
0.2 C
0.2 D
0.35 A
0.25
0.4
0.6
1.0
0
1

Exact Solution Strategies

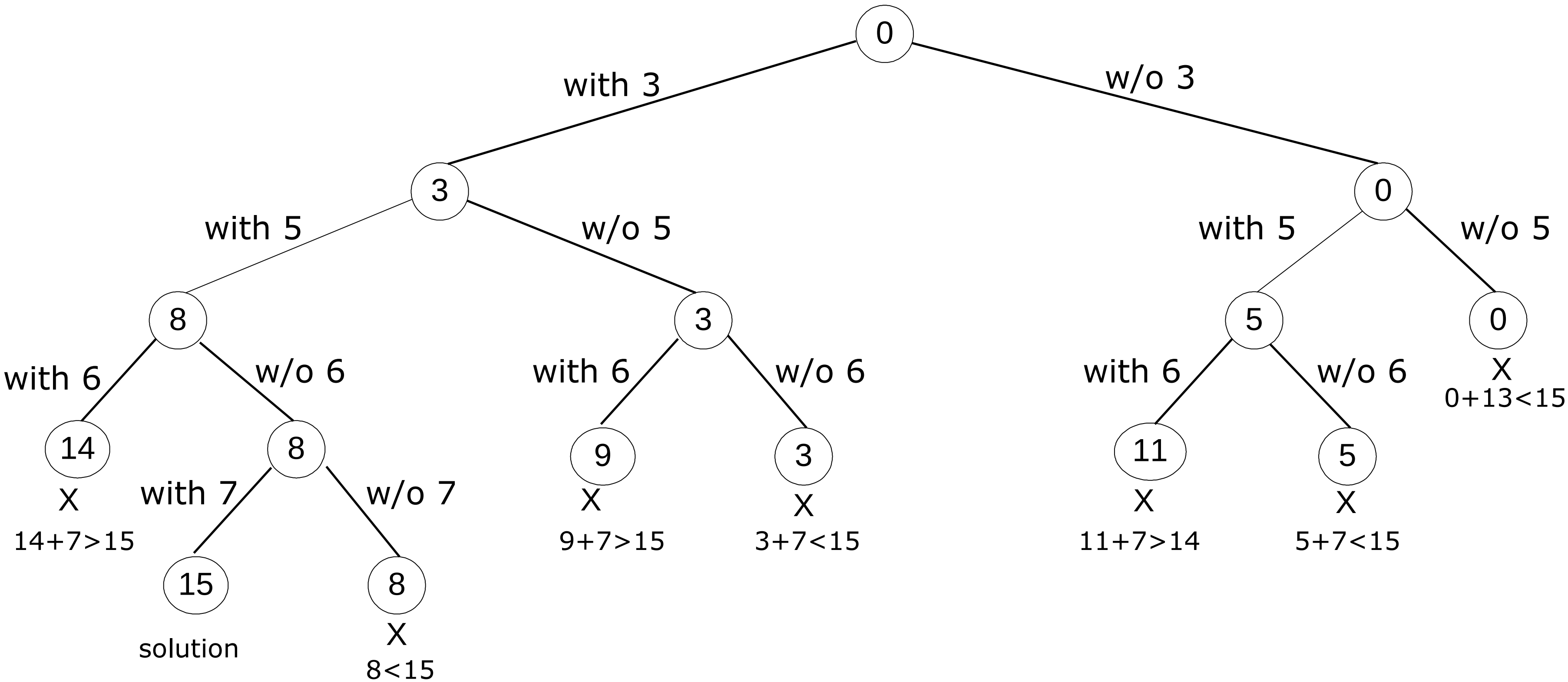
- *exhaustive search* (brute force)
  - useful only for small instances
- *dynamic programming*
  - applicable to some problems (e.g., the knapsack problem)
- *backtracking*
  - eliminates some unnecessary cases from consideration
  - yields solutions in reasonable time for many instances but worst case is still exponential
- *branch-and-bound*
  - further refines the backtracking idea for optimization problems

Backtracking

- Construct the *state-space tree*
  - nodes: partial solutions
  - edges: choices in extending partial solutions
- Explore the state space tree using depth-first search
- “Prune” *nonpromising nodes*
  - dfs stops exploring subtrees rooted at nodes that cannot lead to a solution and backtracks to such a node’s parent to continue the search

Example: Subset-Sum Problem

A={3, 5, 6, 7}   d=15



Branch-and-Bound

- An enhancement of backtracking
- Applicable to optimization problems
- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution)
- Uses the bound for:
  - ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far
  - guiding the search through state-space

Example: Assignment Problem

Select one element in each row of the cost matrix C so that:

- no two selected elements are in the same column
- the sum is minimized

Example

	Job 1	Job 2	Job 3	Job 4
Person *a*	9	2	7	8
Person *b*	6	4	3	7
Person *c*	5	8	1	8
Person *d*	7	6	9	4

<u>Lower bound</u>: Any solution to this problem will have total cost at least: 2 + 3 + 1 + 4 (or 5 + 2 + 1 + 4)

Example: Complete state-space tree

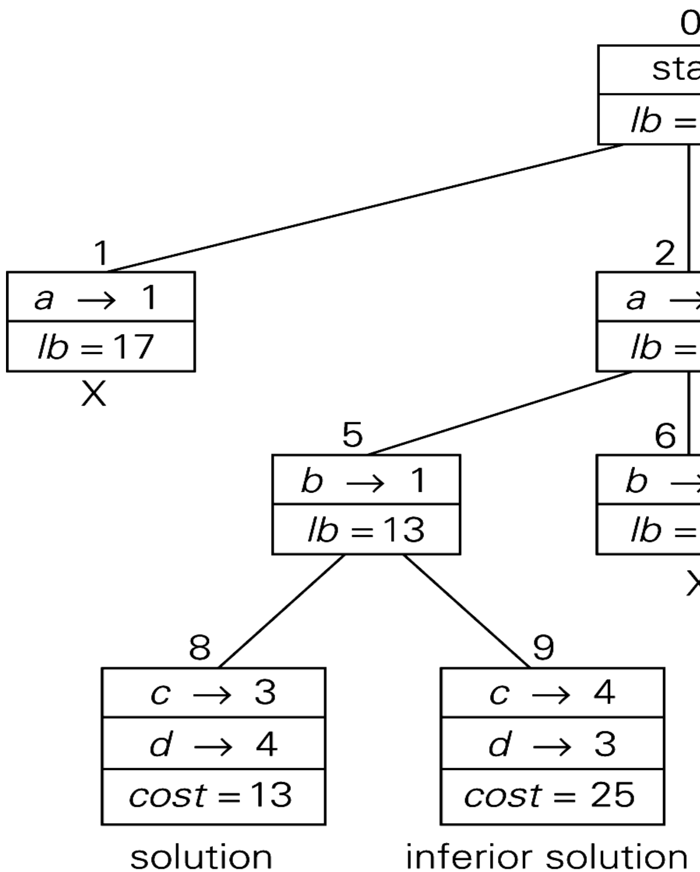
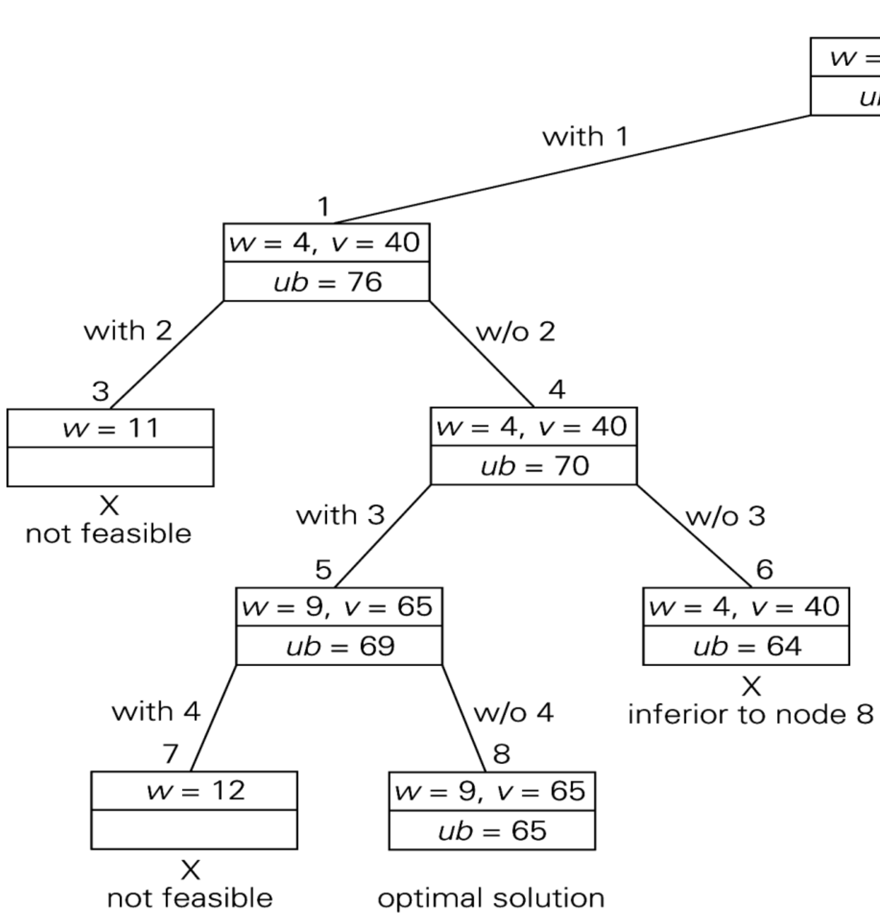


FIGURE 12.7 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm

Example: Knapsack Problem



item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

FIGURE 12.8 State-space tree of the branch-and-bound algorithm for the instance of the knapsack problem

Example: Traveling Salesman Problem

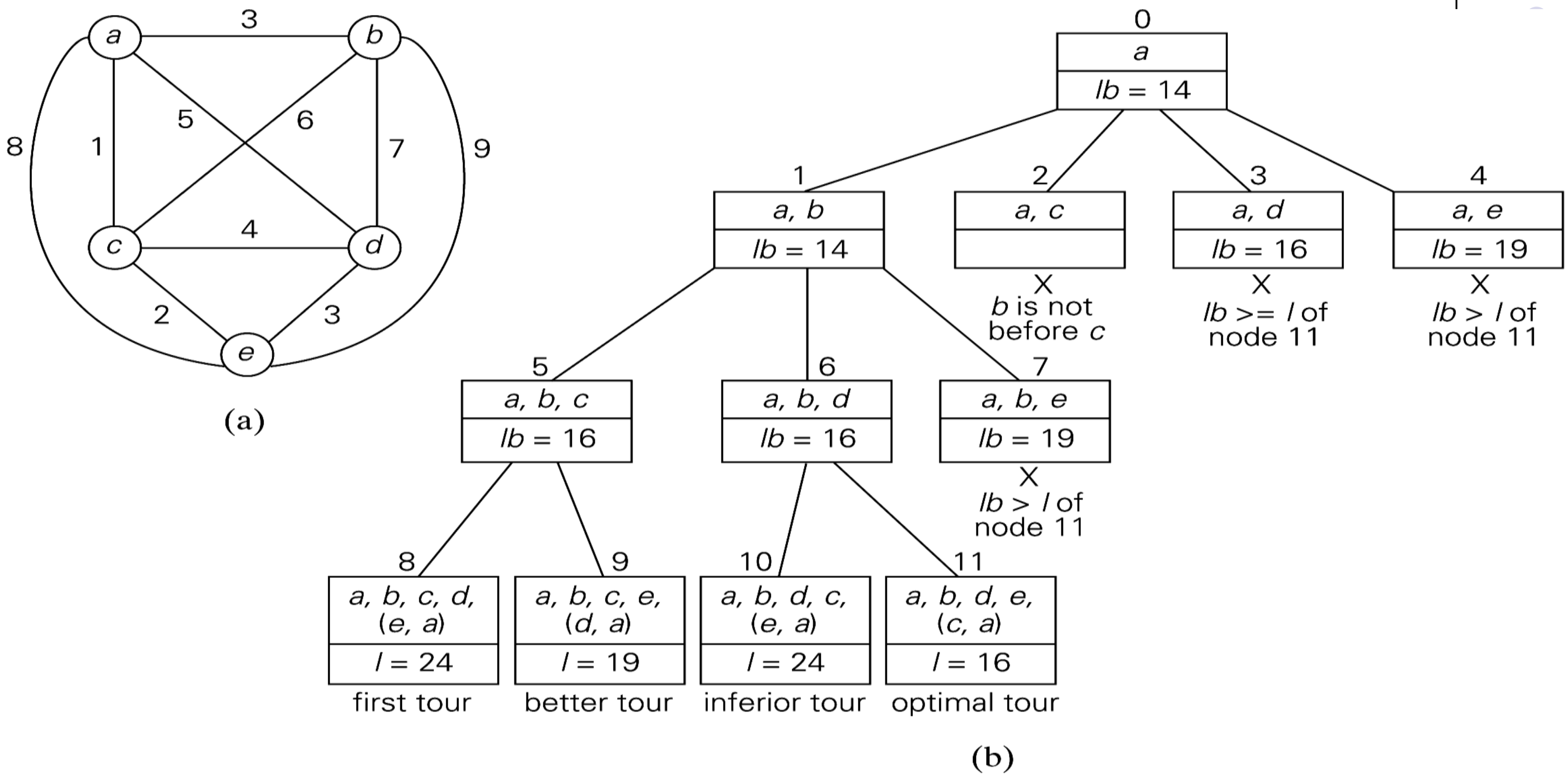


FIGURE 12.9 (a) Weighted graph. (b) State-space tree of the the branch-and-bound algorithm to find the shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.