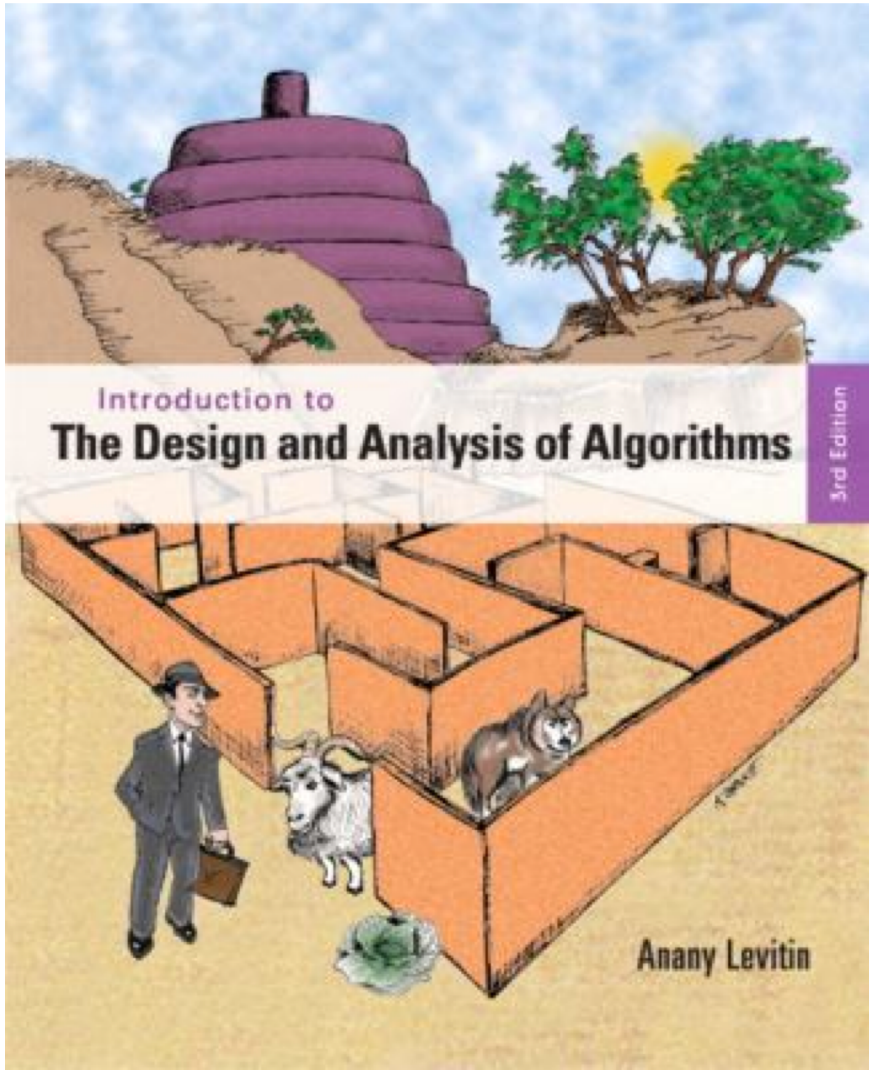
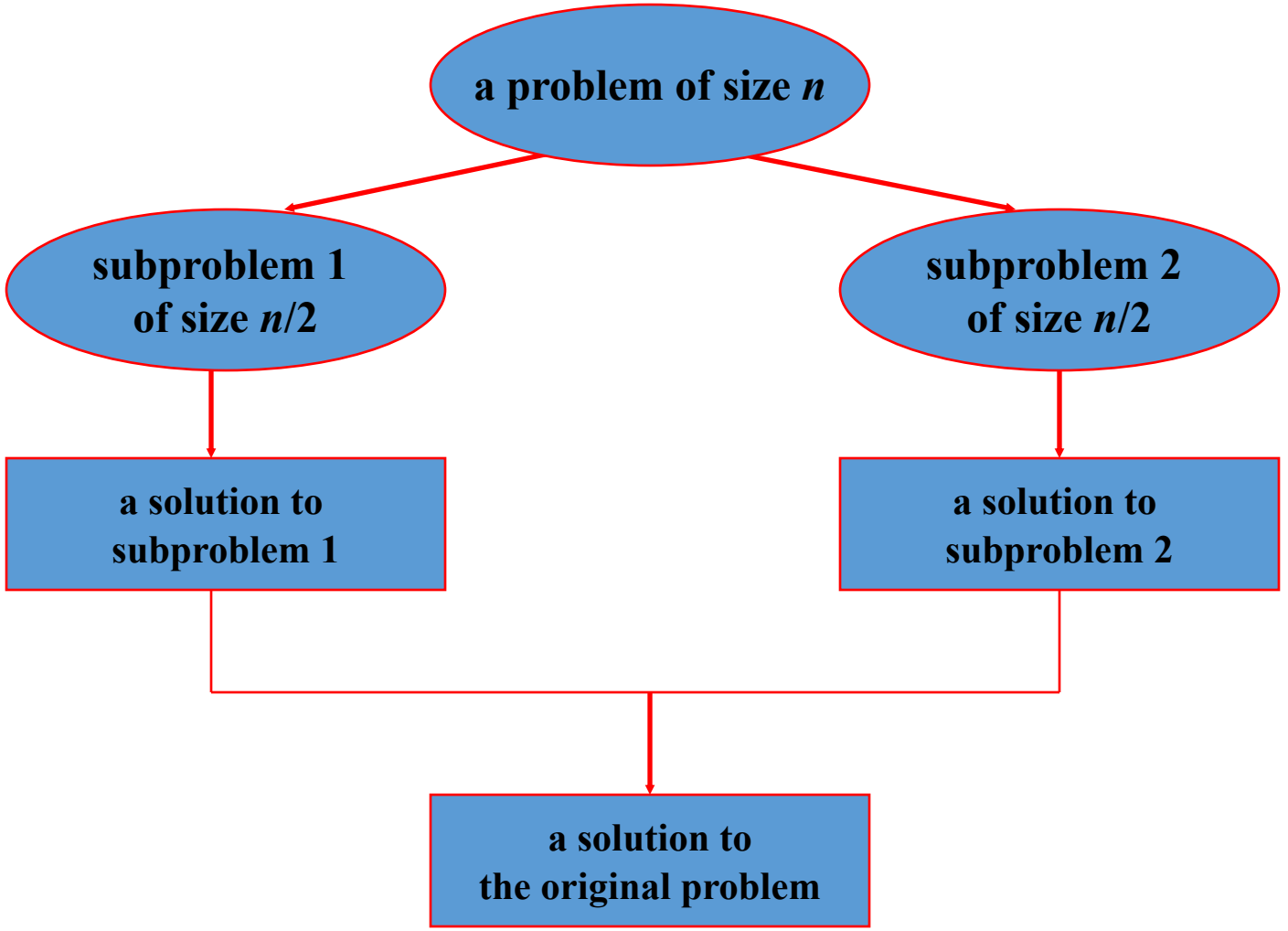
Review of Chapters 5-8



Chapter 5 Divide-and-Conquer

- Solving subproblems recursively, and then combine their solutions to get a solution to the original problem



Running time

$T(n) = aT(n/b) + f(n)$

The *Master Theorem* establishes the order of growth of its solutions.

ALGORITHM *Mergesort*$(A[0..n-1])$

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A)

Analysis of Mergesort

- **All cases** have same efficiency: $\Theta(n \log n)$
- Stability: stable
- Space requirement: $\Theta(n)$ (not in-place)
- Number of comparisons in the worst case is close to theoretical minimum for comparison-based sorting.

Quicksort

Quicksort is a divide-and-conquer sorting algorithm.

- What is the difference between mergesort and quicksort??
- Time efficiency ??
- Stable ??
- Memory ??

Analysis of Quicksort

- Best case: split in the middle — $\Theta(n \log n)$
- Worst case: sorted array! — $\Theta(n^2)$
- Average case: random arrays — $\Theta(n \log n)$
- Not stable.
- Faster than mergesort for sorting randomly ordered arrays of large size

Multiplication of Large Integers

Let $A = A_1A_0$ and $B = B_1B_0$, where A and B are *n*-digit,

A_1, A_0, B_1, B_0 are *n*/2-digit numbers

Using traditional method, we have:

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_0 + A_0 * B_1) \cdot 10^{n/2} + A_0 * B_0$$

Karatsuba algorithm

Let $A = A_1A_0$ and $B = B_1B_0$

$$A * B = A_1 * B_1 \cdot 10^n + (A_1 * B_0 + A_0 * B_1) \cdot 10^{n/2} + A_0 * B_0$$
$$= \quad C_2 \cdot 10^n \quad + \quad C_1 \cdot 10^{n/2} \quad + C_0$$

Where $C_2 = (A_1 * B_1)$,

$C_0 = (A_0 * B_0$

$C_1 = (A_1 + A_0) * (B_1 + B_0) – (C_2 + C_0)$,

Karatsuba algorithm (cont.)

Recurrence for the number of multiplications M(n):

$$M(n) = 3M(n/2), \quad M(1) = 1$$

Solution: $M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}$

Strassen's Matrix Multiplication

Strassen observed [1969] that the product of two matrices can be computed as follows:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$
$$m_2 = (a_{10} + a_{11}) * b_{00},$$
$$m_3 = a_{00} * (b_{01} - b_{11}),$$
$$m_4 = a_{11} * (b_{10} - b_{00}),$$
$$m_5 = (a_{00} + a_{01}) * b_{11},$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$

Matrix Multiplication

The product of two matrices can be computed as follows:

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} * \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

$$C_{00}=A_{00} * B_{00} + A_{01} * B_{10}$$

Analysis of Strassen's Algorithm (cont.)

If n is not a power of 2, matrices can be padded with zeros.

Number of multiplications:

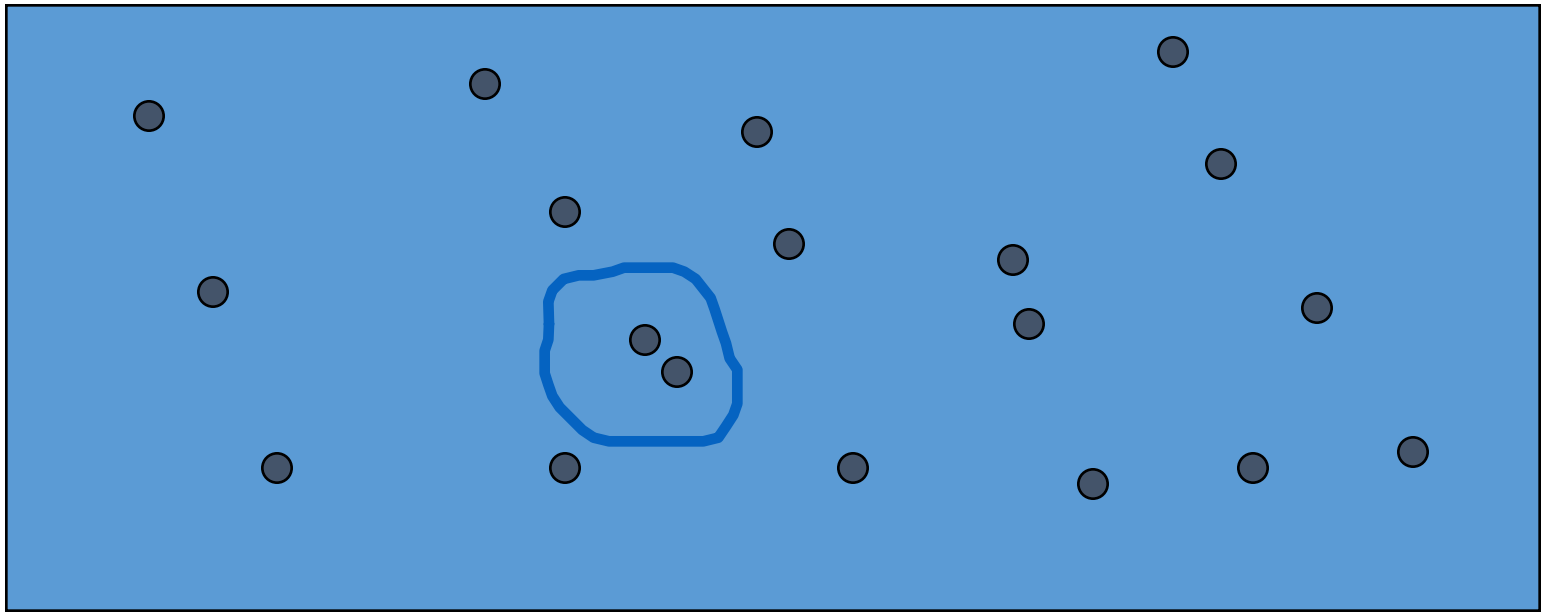
$$M(n) = 7M(n/2) \text{ for } n > 1, \quad M(1) = 1$$

Solution: $M(n) = n^{\log_2 7} \approx n^{2.807}$ vs. n^3 of brute-force alg.

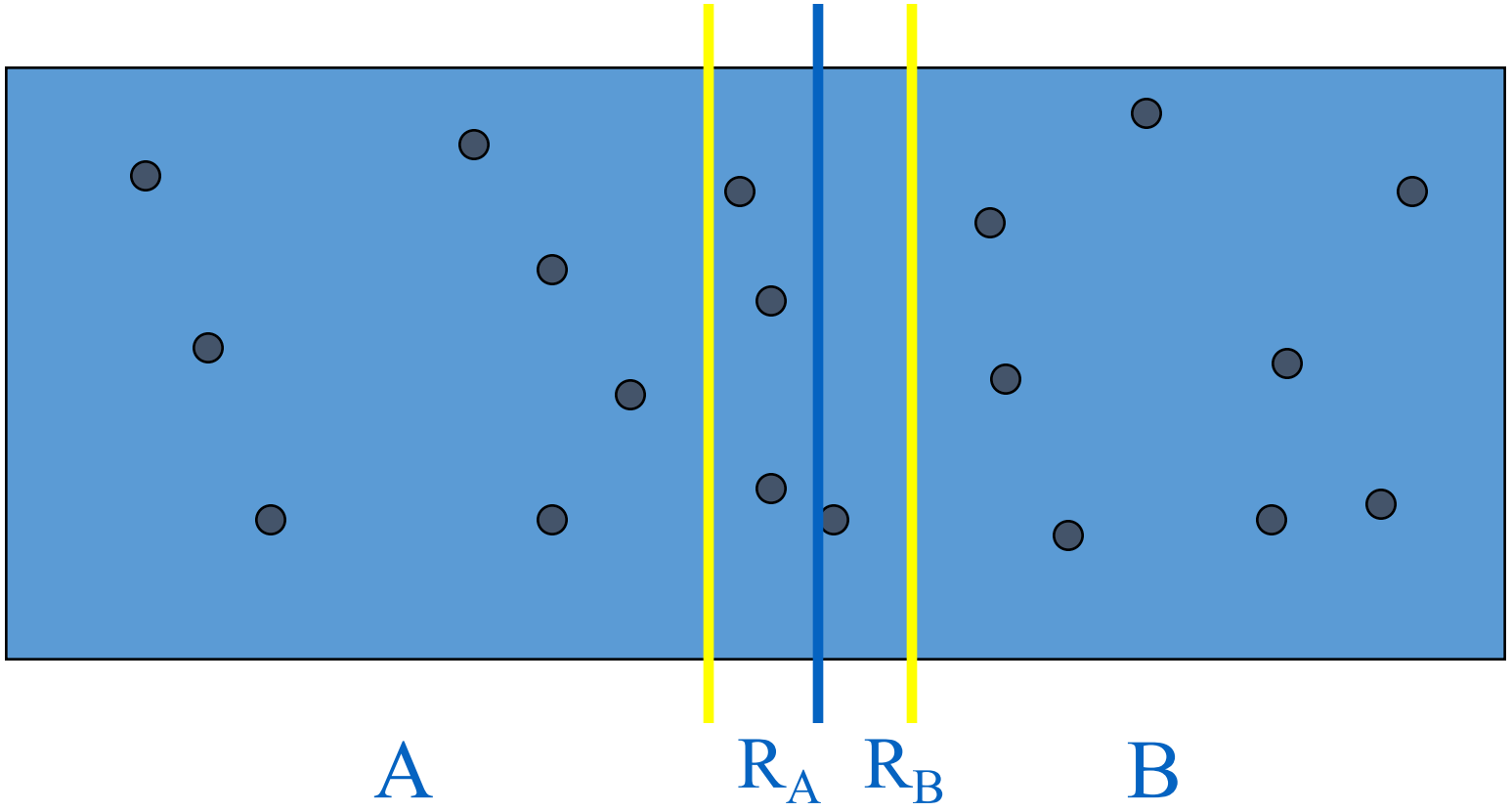
Algorithms with better asymptotic efficiency are known but they are even more complex.

Closest Pair Of Points

- Given *n* points in 2D, find the pair that is closest.



Strategy



- Candidates lie within *d* of the dividing line.
  - Call these regions R_A and R_B, respectively.

Efficiency of the Closest-Pair Algorithm

Running time of the algorithm is described by

$$T(n) = 2T(n/2) + M(n), \text{ where } M(n) \in O(n)$$

By the Master Theorem (with $a = 2$, $b = 2$, $d = 1$)

$$T(n) \in O(n \log n)$$

Other Divide-and-Conquer Algorithms

Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder

Convex Hull Problem

Convex Hull Problem

- Let S be a set of n > 1 points $p_1(x_1, y_1)$, ... , $p_n(x_n, y_n)$ in the Cartesian plane sorted in nondecreasing order of their x coordinates
- Leftmost point p_1 and the rightmost point p_n are two distinct extreme points of the set's convex hull
- Straight line p_1p_n separates the points of S into two sets: S_1 the set of points to the left of this line, and S_2 points to the right of this line.
- Consider two sets separately.

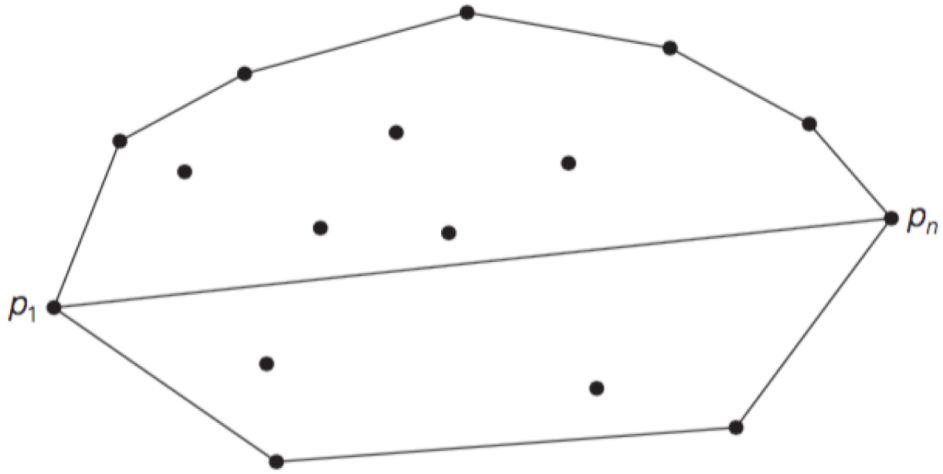
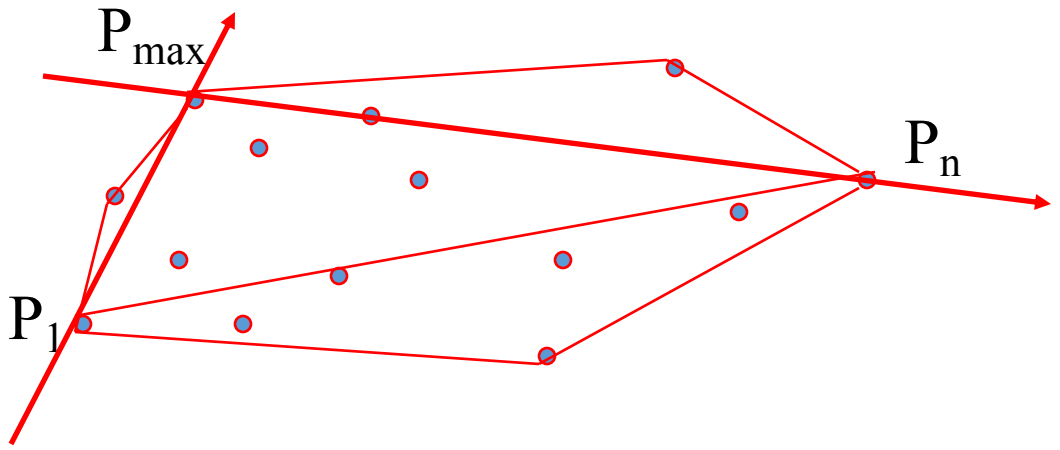


FIGURE 5.8 Upper and lower hulls of a set of points.

Quickhull Algorithm

- Process the points to the left of line P_1P_{max} and points to the left of line $P_{max}P_n$ recursively
- Compute *lower hull* in a similar manner



Efficiency of Quickhull Algorithm

- Finding point farthest away from line P_1P_2 can be done in linear time
- Time efficiency:
  - Best case: $\Theta(n)$
  - worst case: $\Theta(n^2)$ (as quicksort)
  - average case: $\Theta(n)$ (under reasonable assumptions about distribution of points given)
- Several $O(n \log n)$ algorithms for convex hull are known

Chapter 6 Transform and Conquer

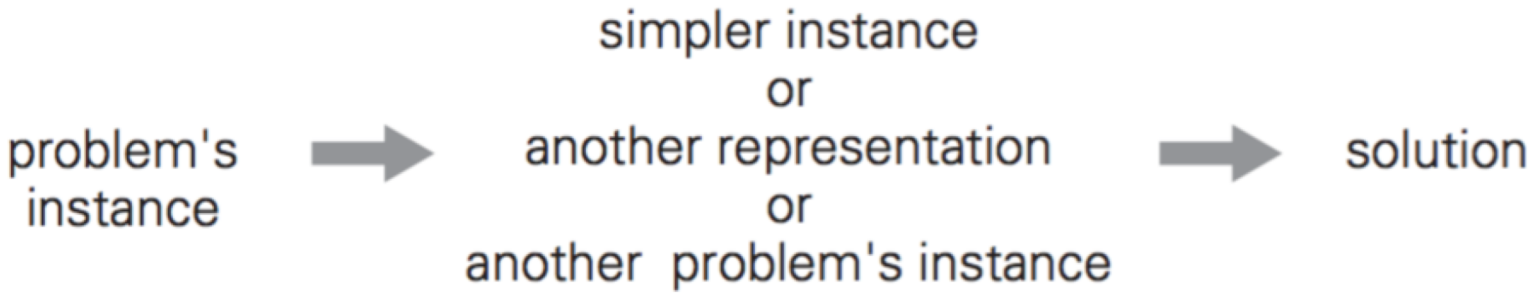


FIGURE 6.1 Transform-and-conquer strategy.

Instance Simplification - Presorting

- **Element uniqueness** problem with presorting
- Presorting-based algorithm

  Stage 1: sort by efficient sorting algorithm (e.g. mergesort)

  Stage 2: scan array to check pairs of adjacent elements

  Efficiency: $\Theta(n \log n) + O(n) = \Theta(n \log n)$

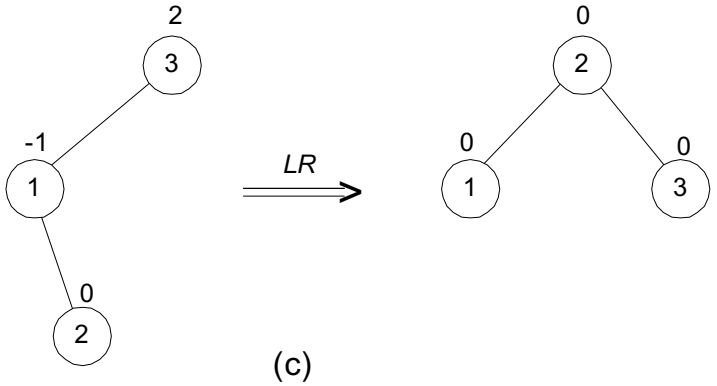
Representation Change – Tree Search

This group of techniques solves a problem by a *transformation* to a different representation of the same instance (*representation change*)

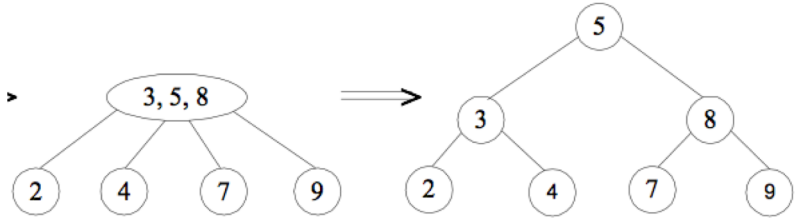
- Example: transformation from a set to a binary search tree
  - ***AVL trees: what is AVL trees***
  - *Multiway search trees*
  - *2-3 trees:* ***what is 2-3 trees?***
  - ***Heaps and heapsort***
  - *B-trees :* ***properties of B-trees?***

AVL trees and *2-3 trees*

- AVL trees
  - How to balance AVL trees?
  - Efficiency of AVL trees
    - Height?
    - Search, insertion, deletion, rotation?



- *2-3 trees*
  - How to balance (2,3) tree?
  - Efficiency of *2-3 trees*
    - Height?
    - Search, insertion, deletion?



Problem Reduction

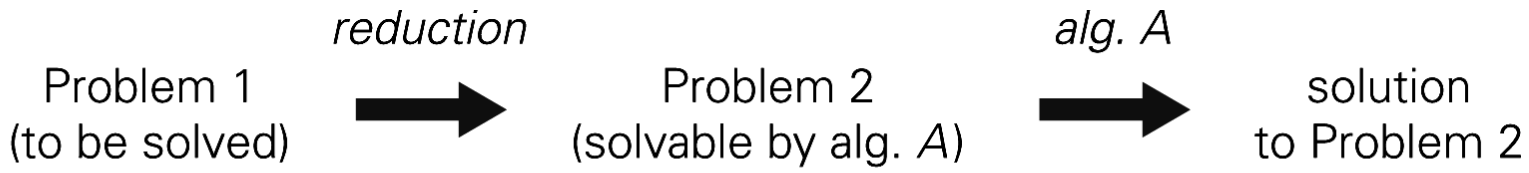


FIGURE 6.15 Problem reduction strategy

- Computing lcm(*m*, *n*) via computing gcd(*m, n*)
- Linear programming
- Reducing puzzle and game problems to standard graph problems

Chapter 7 Space & Time Trade-offs

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: definition
  - string searching algorithms: Horspool's algorithm, Boyer-Moore algorithm
- *Prestructuring* — preprocess the input to make accessing its elements easier
  - hashing
  - indexing schemes (e.g., B-trees)
- *Dynamic programming* (next chapter)

How string searching algorithms shift patterns

- How far to shift?
- Example of Horspool's algorithm

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	_
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

```
BARD LOVED BANANAS
BAOBAB
        BAOBAB
           BAOBAB
                    BAOBAB  (unsuccessful search)
```

Boyer-Moore algorithm

- Which two tables are calculated?
- How to shift pattern when there is a mismatch?
- Time efficiency?

Pre-structuring Technique: Hashing

- What is hashing problem? Three basic operations: find, insert, delete
- Process of hashing?
- Expected time & space efficiency?

Collisions

- What is load factor?
- Strategies to handle collisions: *open hashing (chaining), closed hashing (linear probing, double hashing). Advantage and drawback?*
- Hash table as an array has a fixed size. What is the strategy if it is full?

Chapter 8 Dynamic Programming

- Compare dynamic programming with divide-and-conquer
- Main idea of DP
- Examples
  - Change-Making Problem
  - Calculating Binomial Coefficient
  - Coin-row problem
  - Coin-collecting problem
  - Knapsack Problems
  - Optimal Binary Search Trees: what is optimal binary search trees ?

Coin-row problem

There is a row of n coins whose values are some positive integers $c_1, c_2,\ldots,c_n$, 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.

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 c_n – the max amount is ?
- those with the last coin c_n -- the max amount is ?

Thus we have the following recurrence

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

Change-Making Problem

Give change for amount ***n*** using the minimum number of coins

- For any set of coin denominations
  - Dynamic Programming algorithm: time / space ?
- US coin denominations
  - Greedy algorithm: time / space ?

Dynamic programming algorithm

- Let ***F (n)*** be the minimum number of coins whose values add up to ***n***.

- The amount ***n*** can only be obtained by adding one coin of denomination d_j to the amount $n-d_j$ for $j = 1, 2, \ldots, m$ such that $n > d_j$.

- Consider all such denominations and select the one minimizing $F(n-d_j) + 1$.

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

$$F(0) = 0.$$

Knapsack Problem by DP

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

Let ***F*[*i,j*]** be value of an optimal solution to such instance.

How to derive a recurrence relation that expresses a solution to an instance of the knapsack problem in terms of solutions to its smaller sub-instances ?

Knapsack Problem by DP

- Divide all the subsets of the first i items that fit the knapsack of capacity j into two categories:

  - those that do not include the i^{th} item and
    - the value of an optimal subset is, by definition, $F(i-1, j)$.
  - those that do.
    - The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.

Optimal BST Problem

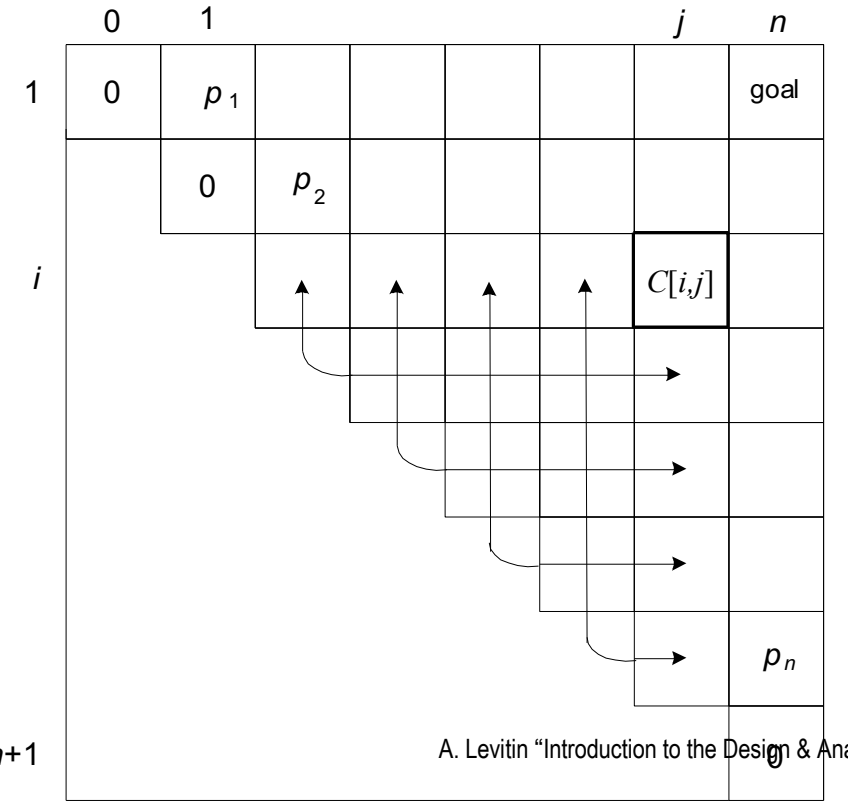
- Given n keys $a_1 < \ldots < a_n$ and probabilities $p_1 , \ldots , p_n$ searching for them,

- Build a BST with a **minimum expected comparisons** in successful search.

DP for Optimal BST Problem (cont.)

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

$$C[i,i] = p_i \quad \text{for } 1 \le i \le j \le n$$



Warshall's algorithm

- Main idea: a path exists between two vertices *i*, *j*, iff
  - there is an edge from *i* to *j*; or
  - there is a path from *i* to *j* going through vertex 1; or
  - there is a path from *i* to *j* going through vertex 1 and/or 2; or
  - …
  - there is a path from *i* to *j* going through vertex 1, 2, … and/or *k*; or
  - …
  - there is a path from *i* to *j* going through any of the other vertices

Construction of Transitive Closure

- Constructed through a series of *nxn* Boolean matrices $R^{(0)}$, …, $R^{(k-1)}$, $R^{(k)}$, …, $R^{(n)}$
- $r_{ij}^{(k)} \in R^{(k)}$, (k=0, 1, …, n) iff $\exists$ a path (of positive length) from i to j with each vertex numbered not higher than k
- The series starts with $R^{(0)}$
- $R^{(1)}$ has information on paths that can use 1 as intermediate, i.e., it may have more 1's than $R^{(0)}$
- $R^{(n)}$ reflects paths that can use all n vertices as intermediate, hence, transitive closure

Recurrence relating elements $R^{(k)}$ to elements of $R^{(k-1)}$

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right)$$

- If $r_{ij} = 1$ in $R^{(k-1)}$, it remains 1 in $R^{(k)}$
- If $r_{ij} = 0$ in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ iff the element in its row i and column k and the element in its column j and row k are both 1's in $R^{(k-1)}$

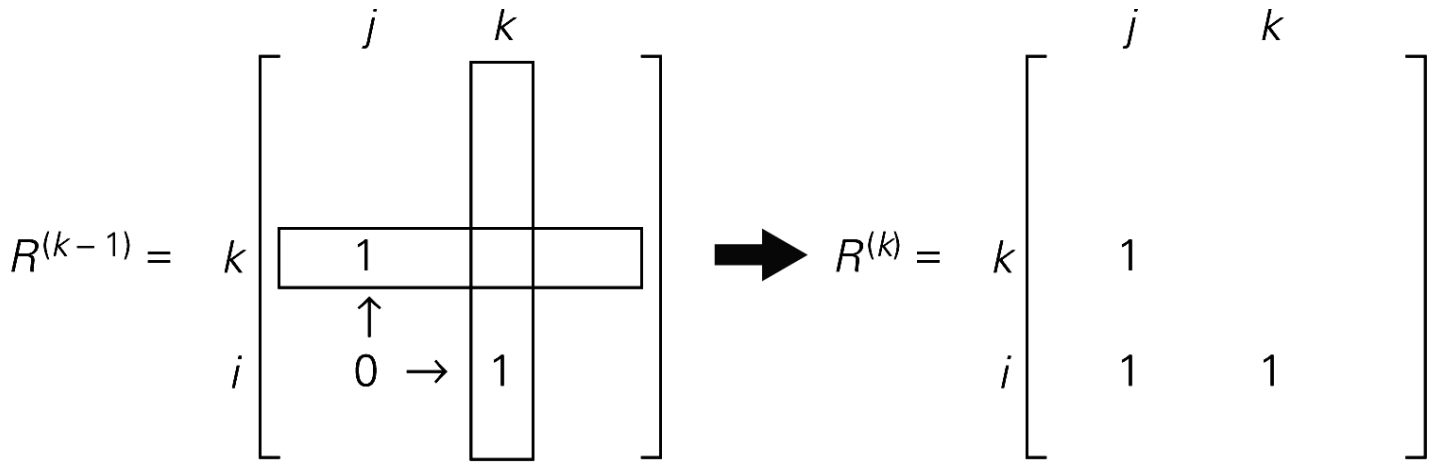


FIGURE 8.3 Rule for changing zeros in Warshall's algorithm

Potential exam questions

Exam covers chapters 5-8

Part I

- 5 multichoice questions
- 5 True / False questions

Part II

- Four questions, one from each chapter
- For dynamic programming question
  - Set up a recurrence relation
  - Based on the recurrence, complete algorithm
  - Describe how to find optimal solution
  - Estimate the efficiency of the algorithm