DATA STRUCTURES AND ALGORITHMS MADE EASY

Data Structures and Algorithmic Puzzles

5TH EDITION

CareerMonk
Together we are better..

Narasimha Karumanchi, M.Tech, IIT Bombay
Founder, CareerMonk.com

Data Structures And Algorithms Made Easy

-To All My Readers

By
Narasimha Karumanchi

☼ Concepts ☼ Problems ☼ Interview Questions

Designed by *Narasimha Karumanchi*

Acknowledgements

Mother and *Father*, it is impossible to thank you adequately for everything you have done, from loving me unconditionally to raising me in a stable household, where your persistent efforts and traditional values taught your children to celebrate and embrace life. I could not have asked for better parents or role-models. You showed me that anything is possible with faith, hard work and determination.

This book would not have been possible without the help of many people. I would like to express my gratitude to all of the people who provided support, talked things over, read, wrote, offered comments, allowed me to quote their remarks and assisted in the editing, proofreading and design. In particular, I would like to thank the following individuals:

- *Mohan Mullapudi*, IIT Bombay, Architect, dataRPM Pvt. Ltd.
- *Navin Kumar Jaiswal*, Senior Consultant, Juniper Networks Inc.
- *A. Vamshi Krishna*, IIT Kanpur, Mentor Graphics Inc.
- *Cathy Reed, BA, MA*, Copy Editor

–*Narasimha Karumanchi*
M-Tech, *IIT Bombay*
Founder, *CareerMonk.com*

Preface

Dear Reader,

Please hold on! I know many people typically do not read the Preface of a book. But I strongly recommend that you read this particular Preface.

It is not the main objective of this book to present you with the theorems and proofs on *data structures* and *algorithms*. I have followed a pattern of improving the problem solutions with different complexities (for each problem, you will find multiple solutions with different, and reduced, complexities). Basically, it's an enumeration of possible solutions. With this approach, even if you get a new question, it will show you a way to *think* about the possible solutions. You will find this book useful for interview preparation, competitive exams preparation, and campus interview preparations.

As a *job seeker*, if you read the complete book, I am sure you will be able to challenge the interviewers. If you read it as an *instructor*, it will help you to deliver lectures with an approach that is easy to follow, and as a result your students will appreciate the fact that they have opted for Computer Science / Information Technology as their degree.

This book is also useful for *Engineering degree students* and *Masters degree students* during their academic preparations. In all the chapters you will see that there is more emphasis on problems and their analysis rather than on theory. In each chapter, you will first read about the basic required theory, which is then followed by a section on problem sets. In total, there are approximately 700 algorithmic problems, all with solutions.

If you read the book as a *student* preparing for competitive exams for Computer Science / Information Technology, the content covers *all the required topics* in full detail. While writing this book, my main focus was to help students who are preparing for these exams.

In all the chapters you will see more emphasis on problems and analysis rather than on theory. In each chapter, you will first see the basic required theory followed by various problems.

For many problems, *multiple* solutions are provided with different levels of complexity. We start with the *brute force* solution and slowly move toward the *best solution* possible for that problem. For each problem, we endeavor to understand how much time the algorithm takes and how much memory the algorithm uses.

It is recommended that the reader does at least one *complete* reading of this book to gain a full understanding of all the topics that are covered. Then, in subsequent readings you can skip directly to any chapter to refer to a specific topic. Even though many readings have been done for the purpose of correcting errors, there could still be some minor typos in the book. If any are found, they will be updated at *www.CareerMonk.com*. You can monitor this site for any corrections and also for new problems and solutions. Also, please provide your valuable suggestions at: *Info@CareerMonk.com*.

I wish you all the best and I am confident that you will find this book useful.

–Narasimha Karumanchi
M-Tech, *I IT Bombay*
Founder, *CareerMonk.com*

Other Books by Narasimha Karumanchi

- **IT Interview Questions**
- **Data Structures and Algorithms for GATE**
- **Data Structures and Aigorithms Made Easy in Java**
- **Coding Interview Questions**
- **Peeling Design Patterns**
- **Elements of Computer Networking**
- **Data Structures and Algorithmic Thinking with Python**

Table of Contents

1. Introduction
   1.1 Variables
   1.2 Data Types
   1.3 Data Structures
   1.4 Abstract Data Types (ADTs)
   1.5 What is an Algorithm?
   1.6 Why the Analysis of Algorithms?
   1.7 Goal of the Analysis of Algorithms
   1.8 What is Running Time Analysis?
   1.9 How to Compare Algorithms
   1.10 What is Rate of Growth?
   1.11 Commonly Used Rates of Growth
   1.12 Types of Analysis
   1.13 Asymptotic Notation
   1.14 Big-O Notation [Upper Bounding Function]
   1.15 Omega-Q Notation [Lower Bounding Function]
   1.16 Theta-Θ Notation [Order Function]
   1.17 Important Notes
   1.18 Why is it called Asymptotic Analysis?
   1.19 Guidelines for Asymptotic Analysis
   1.20 Simplyfying properties of asymptotic notations
   1.21 Commonly used Logarithms and Summations
   1.22 Master Theorem for Divide and Conquer Recurrences
   1.23 Divide and Conquer Master Theorem: Problems & Solutions
   1.24 Master Theorem for Subtract and Conquer Recurrences
   1.25 Variant of Subtraction and Conquer Master Theorem
   1.26 Method of Guessing and Confirming

1.27 Amortized Analysis
1.28 Algorithms Analysis: Problems & Solutions

2. Recursion and Backtracking
2.1 Introduction
2.2 What is Recursion?
2.3 Why Recursion?
2.4 Format of a Recursive Function
2.5 Recursion and Memory (Visualization)
2.6 Recursion versus Iteration
2.7 Notes on Recursion
2.8 Example Algorithms of Recursion
2.9 Recursion: Problems & Solutions
2.10 What is Backtracking?
2.11 Example Algorithms of Backtracking
2.12 Backtracking: Problems & Solutions

3. Linked Lists
3.1 What is a Linked List?
3.2 Linked Lists ADT
3.3 Why Linked Lists?
3.4 Arrays Overview
3.5 Comparison of Linked Lists with Arrays & Dynamic Arrays
3.6 Singly Linked Lists
3.7 Doubly Linked Lists
3.8 Circular Linked Lists
3.9 A Memory-efficient Doubly Linked List
3.10 Unrolled Linked Lists
3.11 Skip Lists
3.12 Linked Lists: Problems & Solutions

4. Stacks
4.1 What is a Stack?
4.2 How Stacks are used
4.3 Stack ADT

4.4 Applications
4.5 Implementation
4.6 Comparison of Implementations
4.7 Stacks: Problems & Solutions

5. Queues
5.1 What is a Queue?
5.2 How are Queues Used?
5.3 Queue ADT
5.4 Exceptions
5.5 Applications
5.6 Implementation
5.7 Queues: Problems & Solutions

6. Trees
6.1 What is a Tree?
6.2 Glossary
6.3 Binary Trees
6.4 Types of Binary Trees
6.5 Properties of Binary Trees
6.6 Binary Tree Traversals
6.7 Generic Trees (*N*-ary Trees)
6.8 Threaded Binary Tree Traversals (Stack or Queue-less Traversals)
6.9 Expression Trees
6.10 XOR Trees
6.11 Binary Search Trees (BSTs)
6.12 Balanced Binary Search Trees
6.13 AVL (Adelson-Velskii and Landis) Trees
6.14 Other Variations on Trees

7. Priority Queues and Heaps
7.1 What is a Priority Queue?
7.2 Priority Queue ADT
7.3 Priority Queue Applications
7.4 Priority Queue Implementations

7.5 Heaps and Binary Heaps
7.6 Binary Heaps
7.7 Heapsort
7.8 Priority Queues [Heaps]: Problems & Solutions

8. Disjoint Sets ADT
8.1 Introduction
8.2 Equivalence Relations and Equivalence Classes
8.3 Disjoint Sets ADT
8.4 Applications
8.5 Tradeoffs in Implementing Disjoint Sets ADT
8.8 Fast UNION Implementation (Slow FIND)
8.9 Fast UNION Implementations (Quick FIND)
8.10 Summary
8.11 Disjoint Sets: Problems & Solutions

9. Graph Algorithms
9.1 Introduction
9.2 Glossary
9.3 Applications of Graphs
9.4 Graph Representation
9.5 Graph Traversals
9.6 Topological Sort
9.7 Shortest Path Algorithms
9.8 Minimal Spanning Tree
9.9 Graph Algorithms: Problems & Solutions

10. Sorting
10.1 What is Sorting?
10.2 Why is Sorting Necessary?
10.3 Classification of Sorting Algorithms
10.4 Other Classifications
10.5 Bubble Sort
10.6 Selection Sort
10.7 Insertion Sort

10.8 Shell Sort
10.9 Merge Sort
10.10 Heap Sort
10.11 Quick Sort
10.12 Tree Sort
10.13 Comparison of Sorting Algorithms
10.14 Linear Sorting Algorithms
10.15 Counting Sort
10.16 Bucket Sort (or Bin Sort)
10.17 Radix Sort
10.18 Topological Sort
10.19 External Sorting
10.20 Sorting: Problems & Solutions

11. Searching
11.1 What is Searching?
11.2 Why do we need Searching?
11.3 Types of Searching
11.4 Unordered Linear Search
11.5 Sorted/Ordered Linear Search
11.6 Binary Search
11.7 Interpolation Search
11.8 Comparing Basic Searching Algorithms
11.9 Symbol Tables and Hashing
11.10 String Searching Algorithms
11.11 Searching: Problems & Solutions

12. Selection Algorithms [Medians]
12.1 What are Selection Algorithms?
12.2 Selection by Sorting
12.3 Partition-based Selection Algorithm
12.4 Linear Selection Algorithm - Median of Medians Algorithm
12.5 Finding the K Smallest Elements in Sorted Order
12.6 Selection Algorithms: Problems & Solutions

13. Symbol Tables
13.1 Introduction
13.2 What are Symbol Tables?
13.3 Symbol Table Implementations
13.4 Comparison Table of Symbols for Implementations

14. Hashing
14.1 What is Hashing?
14.2 Why Hashing?
14.3 HashTable ADT
14.4 Understanding Hashing
14.5 Components of Hashing
14.6 Hash Table
14.7 Hash Function
14.8 Load Factor
14.9 Collisions
14.10 Collision Resolution Techniques
14.11 Separate Chaining
14.12 Open Addressing
14.13 Comparison of Collision Resolution Techniques
14.14 How Hashing Gets O(1) Complexity?
14.15 Hashing Techniques
14.16 Problems for which Hash Tables are not suitable
14.17 Bloom Filters
14.18 Hashing: Problems & Solutions

15. String Algorithms
15.1 Introduction
15.2 String Matching Algorithms
15.3 Brute Force Method
15.4 Rabin-Karp String Matching Algorithm
15.5 String Matching with Finite Automata
15.6 KMP Algorithm
15.7 Boyer-Moore Algorithm

15.8 Data Structures for Storing Strings
15.9 Hash Tables for Strings
15.10 Binary Search Trees for Strings
15.11 Tries
15.12 Ternary Search Trees
15.13 Comparing BSTs, Tries and TSTs
15.14 Suffix Trees
15.15 String Algorithms: Problems & Solutions

16. Algorithms Design Techniques
16.1 Introduction
16.2 Classification
16.3 Classification by Implementation Method
16.4 Classification by Design Method
16.5 Other Classifications

17. Greedy Algorithms
17.1 Introduction
17.2 Greedy Strategy
17.3 Elements of Greedy Algorithms
17.4 Does Greedy Always Work?
17.5 Advantages and Disadvantages of Greedy Method
17.6 Greedy Applications
17.7 Understanding Greedy Technique
17.8 Greedy Algorithms: Problems & Solutions

18. Divide and Conquer Algorithms
18.1 Introduction
18.2 What is the Divide and Conquer Strategy?
18.3 Does Divide and Conquer Always Work?
18.4 Divide and Conquer Visualization
18.5 Understanding Divide and Conquer
18.6 Advantages of Divide and Conquer
18.7 Disadvantages of Divide and Conquer
18.8 Master Theorem

18.9 Divide and Conquer Applications

18.10 Divide and Conquer: Problems & Solutions

19. Dynamic Programming

19.1 Introduction

19.2 What is Dynamic Programming Strategy?

19.3 Properties of Dynamic Programming Strategy

19.4 Can Dynamic Programming Solve All Problems?

19.5 Dynamic Programming Approaches

19.6 Examples of Dynamic Programming Algorithms

19.7 Understanding Dynamic Programming

19.8 Longest Common Subsequence

19.9 Dynamic Programming: Problems & Solutions

20. Complexity Classes

20.1 Introduction

20.2 Polynomial/Exponential Time

20.3 What is a Decision Problem?

20.4 Decision Procedure

20.5 What is a Complexity Class?

20.6 Types of Complexity Classes

20.7 Reductions

20.8 Complexity Classes: Problems & Solutions

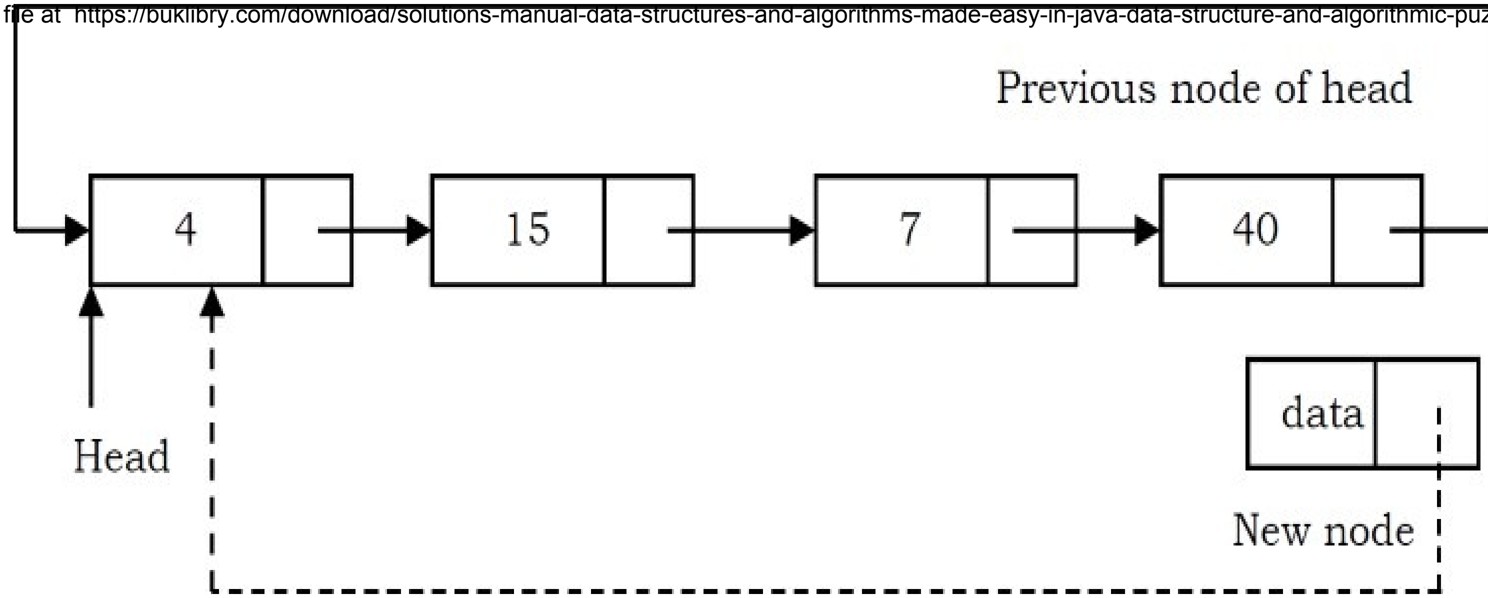
21. Miscellaneous Concepts

21.1 Introduction

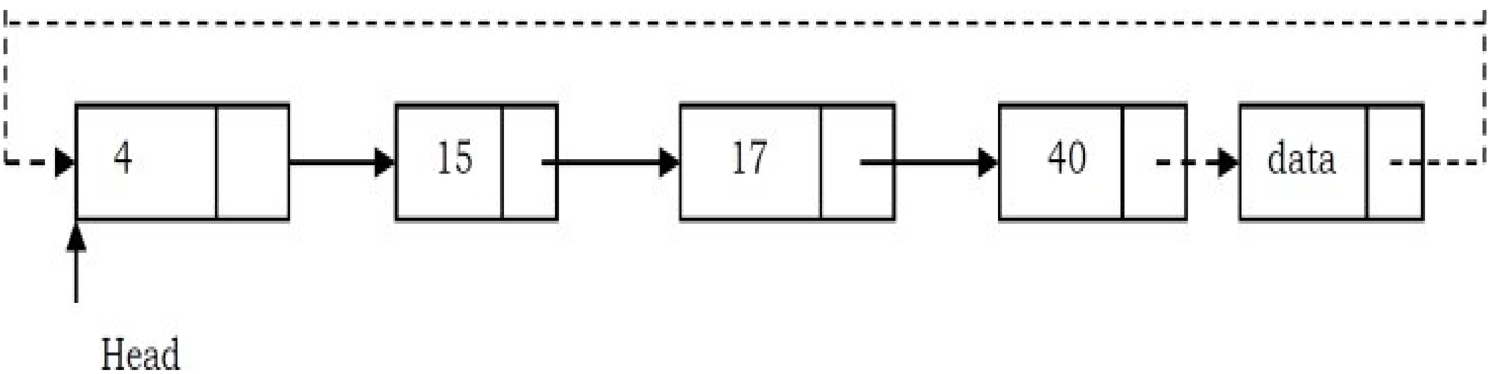
21.2 Hacks on Bit-wise Programming

21.3 Other Programming Questions

References



- Update the next pointer of the previous node to point to the new node and we get the list as shown below.



```
struct BinaryTreeNode *DeepestNodeinBinaryTree(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q;
    if(!root) return NULL;
    Q = CreateQueue();
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        if(temp→left)
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
    return temp;
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-13 Give an algorithm for deleting an element (assuming data is given) from binary tree.

Solution: The deletion of a node in binary tree can be implemented as

- Starting at root, find the node which we want to delete.
- Find the deepest node in the tree.
- Replace the deepest node's data with node to be deleted.
- Then delete the deepest node.

Problem-14 Give an algorithm for finding the number of leaves in the binary tree without using recursion.

Solution: The set of nodes whose both left and right children are NULL are called leaf nodes.

one position. For example: The array is [1 3 -1 -3 5 3 6 7], and w is 3.

Window position	Max
[1 3 -1] -3 5 3 6 7	3
1 [3 -1 -3] 5 3 6 7	3
1 3 [-1 -3 5] 3 6 7	5
1 3 -1 [-3 5 3] 6 7	5
1 3 -1 -3 [5 3 6] 7	6
1 3 -1 -3 5 [3 6 7]	7

Input: A long array A[], and a window width *w*. **Output:** An array B[], B[i] is the maximum value of from A[i] to A[i+w-1]
Requirement: Find a good optimal way to get B[i]

Solution: Brute force solution is, every time the window is moved we can search for a total of *w* elements in the window.

Time complexity: O(*nw*).

Problem-29 For Problem-28, can we reduce the complexity?

Solution: Yes, we can use heap data structure. This reduces the time complexity to O(*nlogw*). Insert operation takes O(*logw*) time, where *w* is the size of the heap. However, getting the maximum value is cheap; it merely takes constant time as the maximum value is always kept in the root (head) of the heap. As the window slides to the right, some elements in the heap might not be valid anymore (range is outside of the current window). How should we remove them? We would need to be somewhat careful here. Since we only remove elements that are out of the window's range, we would need to keep track of the elements' indices too.

Problem-30 For Problem-28, can we further reduce the complexity?

Solution: Yes, The double-ended queue is the perfect data structure for this problem. It supports insertion/deletion from the front and back. The trick is to find a way such that the largest element in the window would always appear in the front of the queue. How would you maintain this requirement as you push and pop elements in and out of the queue?

Besides, you will notice that there are some redundant elements in the queue that we shouldn't even consider. For example, if the current queue has the elements: [10 5 3], and a new element in the window has the element 11. Now, we could have emptied the queue without considering elements 10, 5, and 3, and insert only element 11 into the queue.

- For each element of the input array, insert into the hash table. Let us say the current element is $A[X]$.
- Check whether there exists a hash entry in the table with key: $K - A[X]$.
- If such element exists then scan the element pairs of $K - A[X]$ and return all possible pairs by including $A[X]$ also.
- If no such element exists (with $K - A[X]$ as key) then go to next element.

Time Complexity: The time for storing all possible pairs in Hash table + searching = $O(n^2) + O(n^2) \approx O(n^2)$. Space Complexity: $O(n)$.

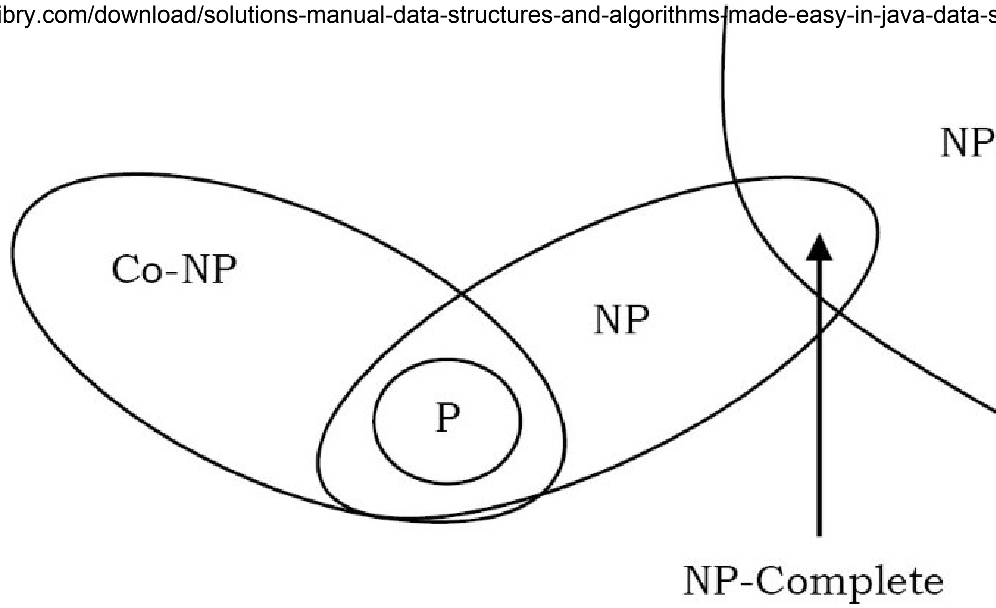
Problem-36 Given an array of n integers, the 3 – *sum problem* is to find three integers whose sum is closest to *zero*.

Solution: This is the same as that of Problem-32 with K value is zero.

Problem-37 Let A be an array of n distinct integers. Suppose A has the following property: there exists an index $1 \le k \le n$ such that $A[1],..., A[k]$ is an increasing sequence and $A[k + 1],..., A[n]$ is a decreasing sequence. Design and analyze an efficient algorithm for finding k.

Similar question: Let us assume that the given array is sorted but starts with negative numbers and ends with positive numbers [such functions are called monotonically increasing functions]. In this array find the starting index of the positive numbers. Assume that we know the length of the input array. Design a $O(logn)$ algorithm.

Solution: Let us use a variant of the binary search.



The set of problems that are *NP*-hard is a strict superset of the problems that are *NP*-complete. Some problems (like the halting problem) are *NP*-hard, but not in *NP*. *NP-hard* problems might be impossible to solve in general. We can tell the difference in difficulty between *NP*-hard and *NP*-complete problems because the class *NP* includes everything easier than its "toughest" problems - if a problem is not in *NP*, it is harder than all the problems in *NP*.

Does P==NP?

If *P* = *NP*, it means that every problem that can be checked quickly can be solved quickly (remember the difference between checking if an answer is right and actually solving a problem).

This is a big question (and nobody knows the answer), because right now there are lots of *NP*-complete problems that can't be solved quickly. If *P* = *NP*, that means there is a way to solve them fast. Remember that "quickly" means not trial-and-error. It could take a billion years, but as long as we didn't use trial and error, it was quick. In future, a computer will be able to change that billion years into a few minutes.

20.7 Reductions

Before discussing reductions, let us consider the following scenario. Assume that we want to solve problem *X* but feel it's very complicated. In this case what do we do?

The first thing that comes to mind is, if we have a similar problem to that of *X* (let us say *Y*), then we try to map *X* to *Y* and use *Y's* solution to solve *X* also. This process is called reduction.