Chapter 1: Introduction

Progress is possible only if we train ourselves to think about programs without thinking of them as pieces of executable code. (Edsger Dijkstra)

What is an Algorithm?

Definition of an Algorithm

An *algorithm* is a sequence of unambiguous instructions for solving a problem in a finite amount of time. An input to an algorithm is an *instance* of the problem.

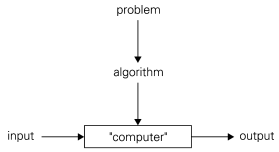


FIGURE 1.1 Notion of algorithm

Example: Euclid's Algorithm

Euclid's algorithm solves the problem of finding the greatest common divisor of two positive integers.

- The allowed inputs and desired output of the problem must be specified.
- Each step in the algorithm must be unambiguous.
- The order of steps must be unambiguous.

We will use *pseudocode* in the book's style.

Pseudocode for Euclid's Algorithm

algorithm $Euclid(m, n)$
// Computes gcd(m, n) by Euclid's algorithm
// Input: Two integers $m > 0$ and $n \geq 0$
// Output: Greatest common divisor of m and n
if $n = 0$ **then**
return m
else
return $Euclid(n, m \bmod n)$

Sieve of Eratosthenes

algorithm $Sieve(n)$
// Implements the sieve of Eratosthenes
// Input: An integer $n > 1$
// Output: A list L of all primes $\leq n$
for $p \leftarrow 2$ to n **do** $A[p] \leftarrow$ true
for $p \leftarrow 2$ to $\lfloor \sqrt{n} \rfloor$ **do**
if $A[p]$ **then** // p is prime
add p to the list L
$j \leftarrow p * p$
while $j \leq n$ **do** // eliminate multiples of p
$A[j] \leftarrow$ false
$j \leftarrow j + p$
return L

Design and Analysis Steps

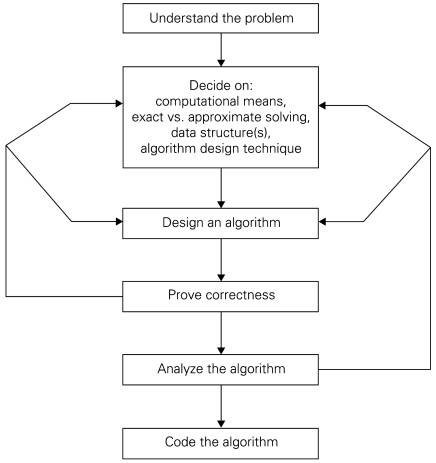


FIGURE 1.2 Algorithm design and analysis process

Algorithm Design Techniques

- Brute Force. Straightforward, naive approach.
- Divide-and-Conquer. Divide into smaller insts.
- Decrease-and-Conquer. Decrease instance size.
- Transform-and-Conquer. Modify instance first.
- Space and Time Tradeoffs. Use more space now to save time later.
- Dynamic Programming. Record results of smaller, reoccuring instances.
- Greedy Technique. Make locally optimal decisions.
- Iterative Improvement. Improve one change at a time.

Important Problem Types

8

Important Problem Types

- Sorting. Arrange items in order.
- Searching. Find items in a data structure.
- String Processing. E.g., string matching.
- Graph Problems. Paths, coloring,
- Combinatorial Problems. Find correct/best combination, permutation, or subset.
- Geometric Problems. Points, lines, shapes, volumes.
- Numerical Problems. Continuous values and models.

Fundamental Data Structures

9

Abstract Data Types

A *data structure* is a way of organizing a group of data items.

A *data type* is a group of data items and the operations defined on them.

An *abstract data type (ADT)* is a data type whose implementation (data structure) is hidden. It can only be accessed or modified via the operations.

For example, a set ADT might be implemented using a hash table.

Linear Data Structures

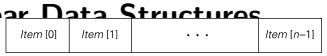


FIGURE 1.3 Array of n elements

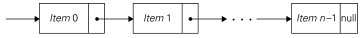


FIGURE 1.4 Singly linked list of n elements

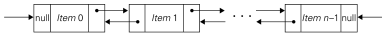


FIGURE 1.5 Doubly linked list of n elements

Stack ADT: push, pop.
Queue ADT: enqueue, dequeue.
Priority Queue ADT: add item, find/delete largest.

Graphs

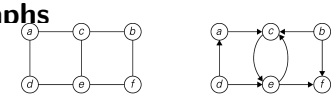


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

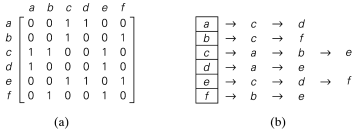


FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a

Trees

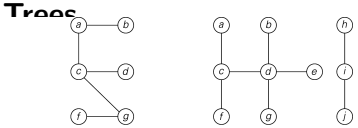


FIGURE 1.10 (a) Tree. (b) Forest.

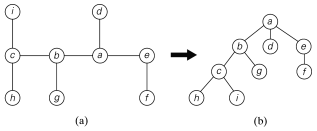


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Binary Search Trees

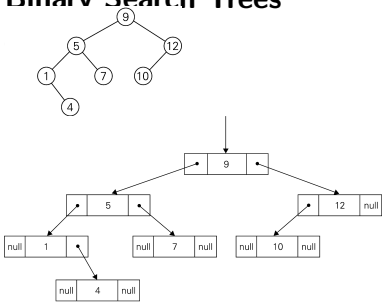
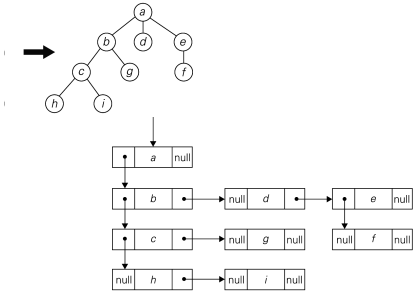


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b

First Child-Next Sibling Representation



Sets and Dictionaries

A *set* is an unordered collection of distinct items.

A *bit vector* can be used to represent a subset of a small set. E.g., 0011010100 might represent the subset $\{2, 3, 5, 7\}$ of $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

A *multiset* or *bag* is an unordered collection of items, not necessarily distinct.

A *list* is an ordered collection of items.

Dictionary ADT: add/search for/delete item.