CS 395 – Analysis of Algorithms

Chapter 1 – Introduction
Read 1.3, 1.4

- **Important problem types**
- **Sieve of Eratosthenes - Primes**
- **Fundamental data structures**
- **Weekly assignment 2**

Fundamentals of Algorithmic Problem Solving

- An algorithm is a sequence of specific instructions for getting an answer.
- Understanding the Problem
  - Trivially must understand the question to be addressed (solved).
  - More importantly what are the properties of the problem domain.
  - Exact range of inputs (instances) over which the algorithm is valid.
- Ascertaining the Capabilities of a Computational Device
  - Computer architecture - {sequential, parallel}

2

Let us start by reiterating an important point made in the introduction to this chapter:

We can consider algorithms to be procedural solutions to problems.

These solutions are not answers but specific instructions for getting answers. It is this emphasis on pre

We now list and briefly discuss a sequence of steps one typically goes through in designing and analyz

Understanding the Problem

From a practical perspective, the first thing you need to do before designing an algorithm is to underst

There are a few types of problems that arise in computing applications quite often. We review them in

An input to an algorithm specifies an ***instance*** of the problem the algorithm solves. It is very importan

Do not skimp on this first step of the algorithmic problem-solving process; otherwise, you will run the

Ascertaining the Capabilities of the Computational Device

Once you completely understand a problem, you need to ascertain the capabilities of the computationa

The central assumption of the RAM model does not hold for some newer computers that can execute o

Should you worry about the speed and amount of memory of a computer at your disposal? If you are d

Dr. BC Note: We will discuss comparison based sorting algorithms later. They are l

Fundamentals of Algorithmic Problem Solving

- Choosing between Exact and Approximate Problem Solving
  - Exact algorithms solve problems exactly.
- Why choose approximate algorithms?
  - Some important problems cannot be solved exactly (e.g., extracting the square roots, solving nonlinear equations, etc).
  - Some problems require prohibitive time to solve exactly. Approximate algorithms can solve problems faster but with less accuracy.
- Deciding on Appropriate Data Structures
  - Many algorithmic solutions require specific data structures for the design of efficient algorithms (e.g., heaps, hash tables, etc.).

3

Choosing between Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximat

In the former case, an algorithm is called an ***exact algorithm***;

in the latter case, an algorithm is called an ***approximation algorithm***.

Why would one opt for an approximation algorithm?

First, there are important problems that simply cannot be solved exactly for most of their instances; ex

Dr. BC Note: Hardware Forwarding Stats

Second, available algorithms for solving a problem exactly can be unacceptably slow because of the p

Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a proble

Fundamentals of Algorithmic Problem Solving

- Algorithm Design Techniques
  - From the book: An algorithm design technique is a general approach to solving problems algorithmically that is applicable to a variety of problems form different areas of computing.
- Methods of Specifying an Algorithm
  - Pseudocode
  - Flowchart
- Proving Algorithm's Correctness
  - Mathematical proof
  - Proof by Mathematical induction (e.g., algorithm must behavior correctly for the simplest problem specification).

4

Algorithm Design Techniques

Now, with all the components of the algorithmic problem solving in place, how do you design an algo

An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problem
Check this book's table of contents and you will see that a majority of its chapters are devoted to indiv
First, they provide guidance for designing algorithms for new problems, i.e., problems for which there
Second, algorithms are the cornerstone of computer science. Every science is interested in classifying

Designing an Algorithm and Data Structures

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic
Of course, one should pay close attention to choosing data structures appropriate for the operations pe

Methods of Specifying an Algorithm

Once you have designed an algorithm, you need to specify it in some fashion. In Section 1.1, to give y
Using a natural language has an obvious appeal; however, the inherent ambiguity of any natural langu
Pseudocode is a mixture of a natural language and programming languagelike constructs. Pseudocode
This book's dialect was selected to cause minimal difficulty for a reader. For the sake of simplicity, we

In the earlier days of computing, the dominant vehicle for specifying algorithms was a ***flowchart***, a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps. This representation technique has proved to be inconvenient for all but very simple algorithms; nowadays, it can be found only in old algorithm books.

The state of the art of computing has not yet reached a point where an algorithm's description—be it in a natural language or pseudocode—can be fed into an electronic computer directly. Instead, it needs to be converted into a computer program written in a particular computer language. We can look at such a program as yet another way of specifying the algorithm, although it is preferable to consider it as the algorithm's implementation.

Proving an Algorithm's Correctness

Once an algorithm has been specified, you have to prove its ***correctness***. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n) (which, in turn, needs a proof; see Problem 7 in Exercises 1.1), the simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs. It might be worth mentioning that although tracing the algorithm's performance for a few specific inputs can be a very worthwhile activity, it cannot prove the algorithm's correctness conclusively. But in order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.

Thenotionofcorrectnessforapproximationalgorithmsislessstraightforward than it is for exact algorithms. For an approximation algorithm, we usually would like to be able to show that the error produced by the algorithm does not exceed a predefined limit. You can find examples of such investigations in Chapter 12.

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency
- Simplicity and Generality
- Does there exist a better algorithm?
  - lower bounds
  - Optimality
- As a rule, a good algorithm is a result of repeated effort and rework

5

Analyzing an Algorithm

We usually want our algorithms to possess several qualities. After correctness, by far the most importa

Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisel

Yet another desirable characteristic of an algorithm is ***generality***. There are, in fact, two issues here: ge

As to the set of inputs, your main concern should be designing an algorithm that can handle a set of in

If you are not satisfied with the algorithm's efficiency, simplicity, or generality, you must return to the

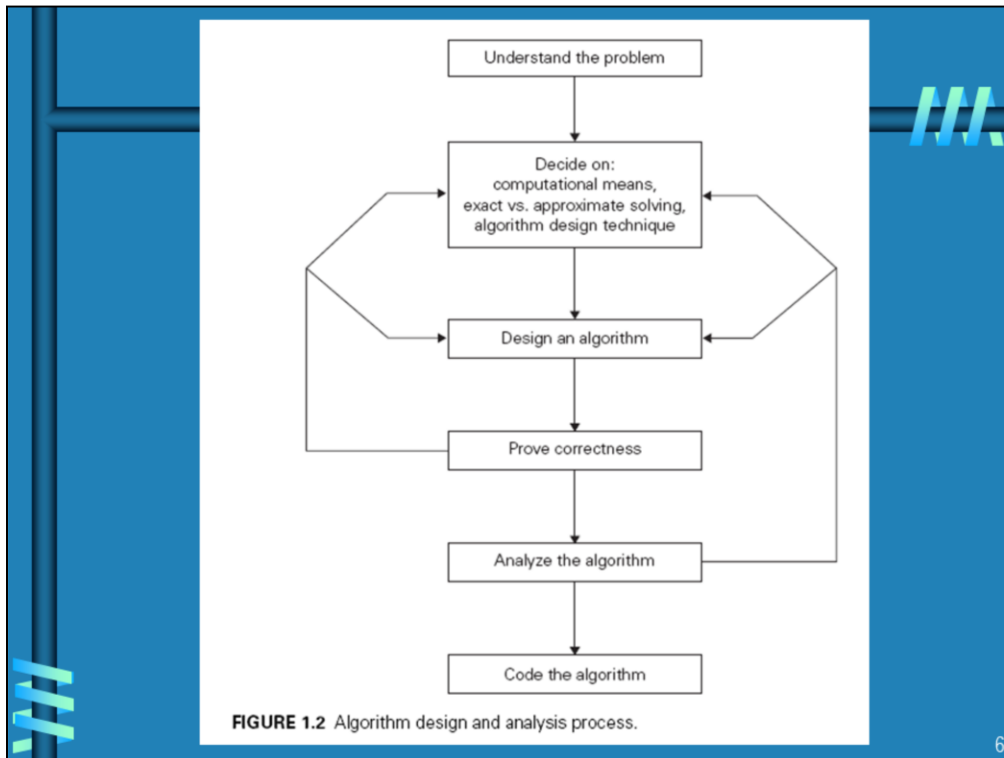


FIGURE 1.2 Algorithm design and analysis process.

(summary of the pervious slides)

…

Coding an Algorithm

Most algorithms are destined to be ultimately implemented as computer programs. Programming an al

As a practical matter, the validity of programs is still established by testing. Testing of computer progr

Also note that throughout the book, we assume that inputs to algorithms belong to the specified sets ar

Of course, implementing an algorithm correctly is necessary but not sufficient: you would not like to d

A working program provides an additional opportunity in allowing an empirical analysis of the underly

In conclusion, let us emphasize again the main lesson of the process depicted in Figure 1.2:

As a rule, a good algorithm is a result of repeated effort and rework.

Even if you have been fortunate enough to get an algorithmic idea that seems perfect, you should still

Actually, this is good news since it makes the ultimate result so much more enjoyable. (Yes, I did think

In the academic world, the question leads to an interesting but usually difficult investigation of an algo

Another important issue of algorithmic problem solving is the question of whether or not every proble

Before leaving this section, let us be sure that you do not have the misconception—possibly caused by the somewhat mechanical nature of the diagram of Figure 1.2—that designing an algorithm is a dull activity. There is nothing further from the truth: inventing (or discovering?) algorithms is a very creative and rewarding process. This book is designed to convince you that this is the case.

Algorithm design techniques/strategies

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

Sieve of Eratosthenes – Find all primes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$

for $p \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**

  if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

    $j \leftarrow p * p$

    while $j \leq n$ **do**

      $A[j] \leftarrow 0$ //mark element as eliminated

      $j \leftarrow j + p$

8

So, let us introduce a simple algorithm for generating consecutive primes not exceeding any given inte

As an example, consider the application of the algorithm to finding the list of primes not exceeding n =

Sieve of Eratosthenes Example

- As an example, consider the application of the algorithm to finding the list of primes not exceeding n = 25:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

2 3 5 7 9 11 13 15 17 19 21 23 25

2 3 5 7 11 13 17 19 23 25

2 3 5 7 11 13 17 19 23

9

For this example, no more passes are needed because they would eliminate numbers already eliminate

What is the largest number p whose multiples can still remain on the list to make further iterations of t

Before we answer this question, let us first note that if p is a number whose multiples are being elimin

Fundamental data structures

- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary

10

Linear data structures

Array/List

FIGURE 1.3 Array of n elements

Linear data structures

Linked List

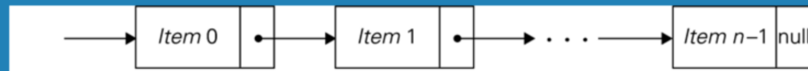
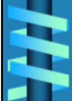


FIGURE 1.4 Singly linked list of n elements



Linear data structures

Doubly-Linked List

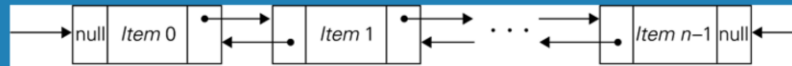


FIGURE 1.5 Doubly linked list of n elements

Graphs

Graphs: $G = <V,E>$

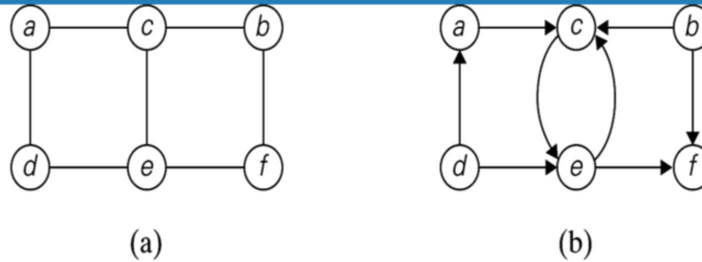


FIGURE 1.6 (a) Undirected graph. (b) Digraph.

14

Graphs

As we mentioned in the previous section, a graph is informally thought of as a collection of points in t

If a pair of vertices (u, v) is not the same as the pair (v, u), we say that the edge (u, v) is ***directed*** from

It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if a

V = {a, b, c, d, e, f}, E = {(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)}.

The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

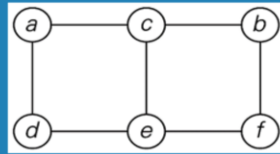
V = {a, b, c, d, e, f}, E = {(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)}.

Our definition of a graph does not forbid ***loops***, or edges connecting vertices to themselves. Unless exp

(We get the largest number of edges in a graph if there is an edge connecting each of its |V| vertices wi

A graph with every pair of its vertices connected by an edge is called ***complete***. A standard notation fo

Graph representations



	a	b	c	d	e	f
a	0	0	1	1	0	0
b	0	0	1	0	0	1
c	1	1	0	0	1	0
d	1	0	0	0	1	0
e	0	0	1	1	0	1
f	0	1	0	0	1	0

(a)

a → c → d
b → c → f
c → a → b → e
d → a → e
e → c → d → f
f → b → e

(b)

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

15

Graph Representations Graphs for computer algorithms are usually represented in one of two ways:

Note that the adjacency matrix of an undirected graph is always symmetric,

i.e., A[i, j]= A[j, i] for every $0 \le i, j \le n - 1$ (why?).

The ***adjacency lists*** of a graph or a digraph is a collection of linked lists, one for each vertex, that cont

If a graph is sparse, the adjacency list representation may use less space than the corresponding adjace

Weighted Graphs

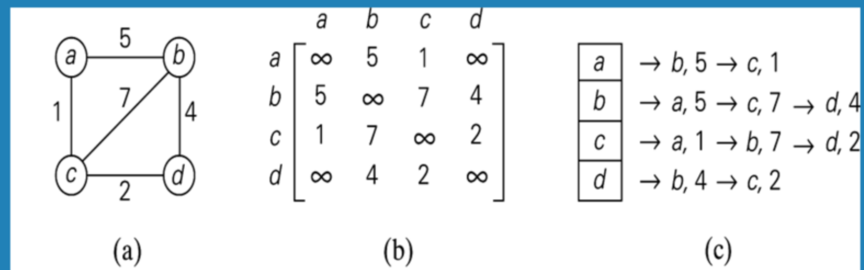


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

16

Weighted Graphs A ***weighted graph*** (or weighted digraph) is a graph (or digraph) with numbers assig
Both principal representations of a graph can be easily adopted to accommodate weighted graphs. If a

Not-connected Graphs

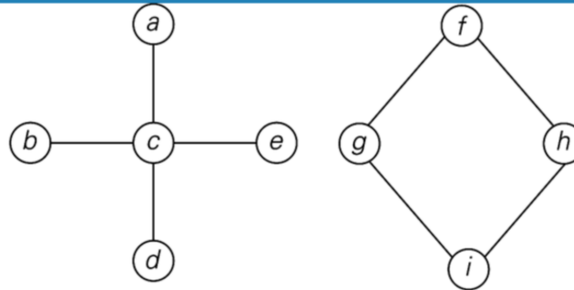
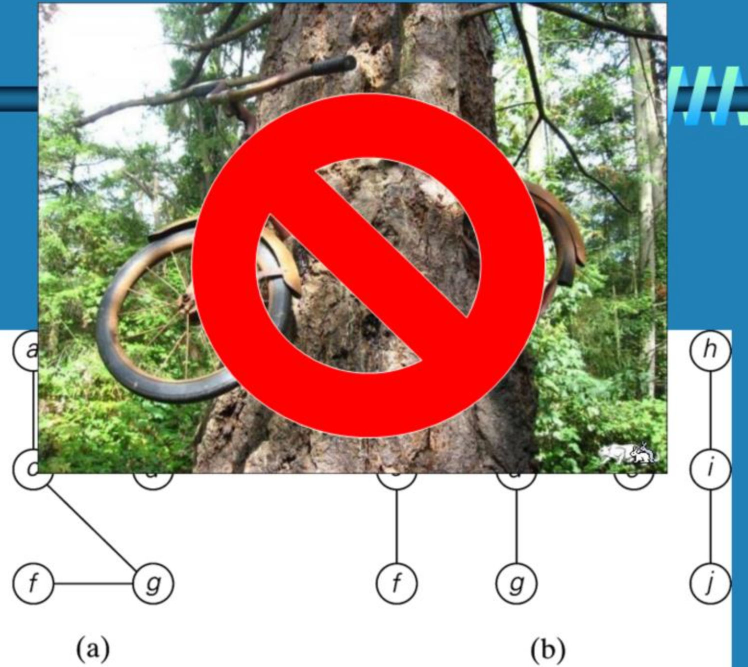


FIGURE 1.9 Graph that is not connected

17

Paths and Cycles Among the many properties of graphs, two are important for a great number of appl
In the case of a directed graph, we are usually interested in directed paths. A ***directed path*** is a sequenc
A graph is said to be ***connected*** if for every pair of its vertices u and v there is a path from u to v. If we
respectively.
Graphs with several connected components do happen in real-world applications. A graph representing
It is important to know for many applications whether or not a graph under consideration has cycles. A



Trees

A ***tree*** (more accurately, a ***free tree***) is a connected acyclic graph (Figure 1.10a). A graph that has no cy

Trees have several important properties other graphs do not have. In particular, th

|E| = |V| − 1.

As the graph in Figure 1.9 demonstrates, this property is necessary but not sufficient for a graph to be

Rooted Trees

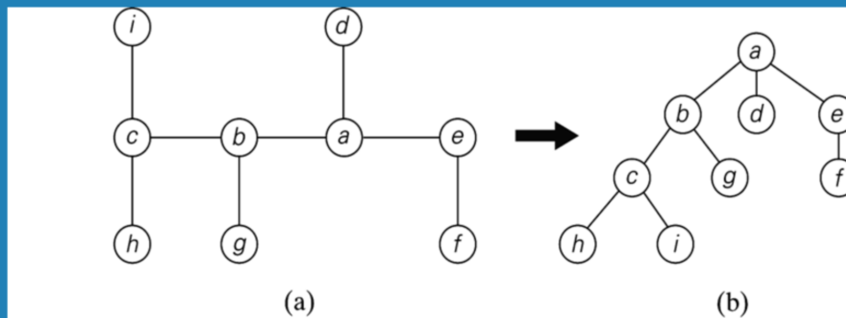


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

19

Rooted Trees Another very important property of trees is the fact **that for every two vertices in**

Dr. BC Note: A simple path has no repeated vertices

Dr. BC Note: A simple graph has no loops or multiple edges

This property makes it possible to select an arbitrary vertex in a free tree and consider it as the ***root*** of

A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacen

Rooted trees play a very important role in computer science, a much more important one than free tree

For any vertex v in a tree T , all the vertices on the simple path from the root to that vertex are called *a*

The ***depth*** of a vertex v is the length of the simple path from the root to v. The ***height*** of a tree is the le

Ordered Trees

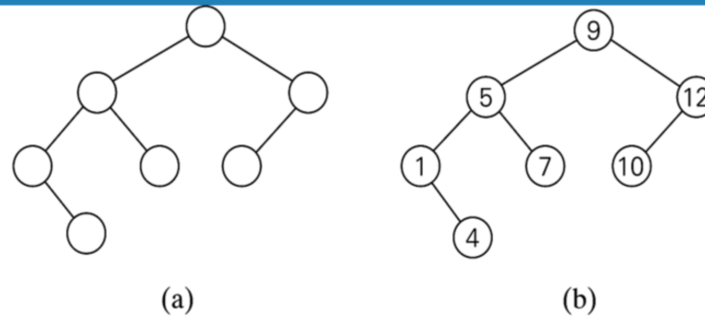


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

20

Ordered Trees An ***ordered tree*** is a rooted tree in which all the children of each vertex are ordered. It

A ***binary tree*** can be defined as an ordered tree in which every vertex has no more than two children a

In Figure 1.12b, some numbers are assigned to vertices of the binary tree in Figure 1.12a. Note that a n

As you will see later in the book, the efficiency of most important algorithms for binary search trees an

A binary tree is usually implemented for computing purposes by a collection of nodes corresponding t

A computer representation of an arbitrary ordered tree can be done by simply providing a parental vert

Binary Search Tree implementation

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

21

… Before next slide:

What if we want multiple (unlimited) children. How could we define that data-structure?

First child-next sibling

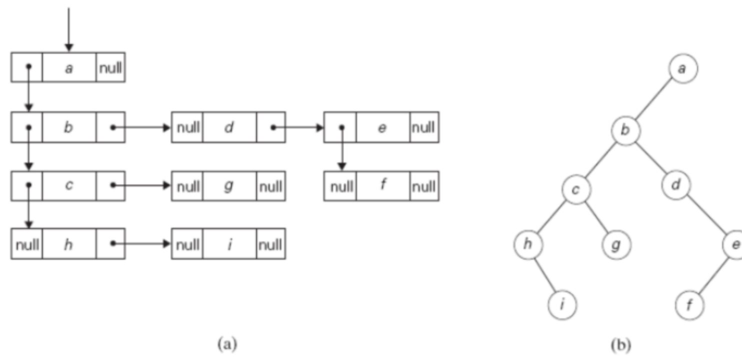
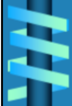


FIGURE 1.14 (a) First child–next sibling representation of the tree in Figure 1.11b. (b) Its binary tree representation.



Sets and Dictionaries

- Set: unordered collection of distinct items called elements.
  - Represented as subset of Universal Set using bit vector
  - Or using Lists
- Multiset / Bag: Allows duplicate elements.
- Dictionary: data structure supporting operations on a set.

23

Sets and Dictionaries

The notion of a set plays a central role in mathematics. A ***set*** can be described as an unordered collecti

Sets can be implemented in computer applications in two ways. The first considers only sets that are s

The second and more common way to represent a set for computing purposes is to use the list structur

In computing, the operations we need to perform for a set or a multiset most often are searching for a

A number of applications in computing require a dynamic partition of some n-element set into a collec

You may have noticed that in our review of basic data structures we almost always mentioned specific

W01 Assignment – lets cut the wire

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

```
//Input: Array A[0..n − 1] of numbers
//Output: Minimum distance between two of its elements
dmin ← ∞
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            if i ≠ j and |A[i] − A[j]| < dmin
            dmin ← |A[i] − A[j]|
return dmin
```

24